Louisiana State University
LSU Digital Commons

LSU Doctoral Dissertations | Graduate School

2017

Empirically Tuning HPC Kernels with iFKO

Md Majedul Haque Sujon
Louisiana State University and Agricultural and Mechanical College, majed.sujon@gmail.com

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Computer Sciences Commons

Recommended Citation

Sujon, Md Majedul Haque, "Empirically Tuning HPC Kernels with iFKO" (2017). *LSU Doctoral Dissertations*. 4427.
https://digitalcommons.lsu.edu/gradschool_dissertations/4427

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contactgradetd@lsu.edu.

EMPIRICALLY TUNING HPC KERNELS WITH IFKO

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The School of Electrical Engineering and Computer Science
The Division of Computer Science and Engineering

by
Md Majedul Haque Sujon
B.S., Bangladesh University of Engineering and Technology, 2005
M.S., University of Texas at San Antonio, 2013
August 2017

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Dr. R. Clint Whaley for the continuous support of my Ph.D. study and research, for his patience, encouragement, and guidance. Besides my advisor, I would like to thank the rest of my dissertation committee: Dr. Jagannathan Ramanujam, Dr. Feng Chen and Dr. Bin Li for their interest in my work. My sincere thanks also goes to Dr. Qing Yi for her advice and encouragement as the Co-PI of one of our projects.

I am thankful to my fellow lab mate Rakib for the stimulating discussions we have had for last couple of years. I am grateful to my parents for supporting me and encouraging me with their best wishes. This dissertation is dedicated to my wife Iva and our little son Yusha. This dissertation would not be possible without their endless sacrifices, supports and warm love.

TABLE OF CONTENTS

ACKNOWLEDGMENTS . . . ii

LIST OF TABLES . . . v

LIST OF FIGURES . . . viii

ABSTRACT . . . ix

CHAPTER 1: INTRODUCTION . . . 1
- 1.1 Terminology and Outline of Research . . . 1
- 1.2 Organization of the Paper . . . 4

CHAPTER 2: TUNING FRAMEWORK . . . 5
- 2.1 iFKO Framework . . . 6
- 2.2 ATLAS Framework . . . 7
- 2.3 Interfacing ATLAS and iFKO . . . 9

CHAPTER 3: ANALYSIS IN FKO TO AID IN SEARCH . . . 10
- 3.1 Architecture Analysis . . . 10
- 3.2 Optloop Analysis . . . 11
- 3.3 Vectorization Analysis . . . 12
- 3.4 Register Spilling Analysis . . . 15

CHAPTER 4: TRANSFORMATIONS ADDED IN FKO TO OPTIMIZE BLAS KERNELS . . . 16
- 4.1 Path Based Optimization . . . 17
- 4.2 SIMD Vectorization . . . 27
- 4.3 SIMD Alignment Issues . . . 34
- 4.4 Architecture Specific Optimization . . . 41
- 4.5 Summary and Conclusions . . . 41

CHAPTER 5: SPECULATIVE VECTORIZATION . . . 42
- 5.1 Description of Speculative Vectorization . . . 46
- 5.2 Integration Within iFKO . . . 55
- 5.3 Experiments . . . 60
- 5.4 Related Work . . . 66
- 5.5 Conclusions and Future Work . . . 67

CHAPTER 6: SLP VECTORIZATION IN FKO . . . 69
- 6.1 Description of SLP in FKO . . . 70
- 6.2 Experiments and Results . . . 89
- 6.3 Related Work . . . 93

6.4 Conclusions and Future Work . . . 94

CHAPTER 7: REPRESENTATION OF TWO DIMENSIONAL ARRAY IN FKO . . . 95
7.1 Declaration of Pointer to Two Dimensional Array in FKO . . . 96
7.2 Experiments and Results . . . 102
7.3 Conclusions and Future Work . . . 104

CHAPTER 8: SUMMARY AND CONCLUSIONS . . . 106

REFERENCES . . . 108

APPENDIX A: ATLAS GEMM MICROKERNELS IN HIL . . . 114
A.1 Double Precision MVEC With $u_m = 12$ $u_n = 4$ $u_k = 1$ . . . 114
A.2 Double Precision KVEC With $u_m = 12$ $u_n = 1$ $u_k = 4$ . . . 119

APPENDIX B: ATLAS GEMM MICROKERNELS IN C . . . 123
B.1 Double Precision MVEC With $u_m = 12$, $u_n = 4$, $u_k = 1$ . . . 123
B.2 Double Precision MVEC With $u_m = 12$, $u_n = 4$, $u_k = 1$ Without Hoisting Loads of pA . . . 125
B.3 Double Precision KVEC Kernel With $u_m = 12$, $u_n = 1$ and $u_k = 4$ . . . 128
B.4 Flags and Pragmas Used to Autovectorize Kernels . . . 129

APPENDIX C: COPYRIGHT PERMISSIONS . . . 131

VITA . . . 132

LIST OF TABLES

3.1 Vectorization related flags of FKO . . . 13

4.1 Vector operations supported in FKO . . . 35

4.2 Examples of the usage of loop markups to handle alignment in FKO. Consider a loop with X, Y, Z arrays and the length of SIMD vector vl bytes . . . 40

5.1 Benchmarks used for experiments . . . 62

5.2 Machine specification . . . 62

6.1 ATLAS's tuning results for *gemm*μ using autovectorization of different compilers 90

6.2 Machine specification . . . 92

7.1 Rows of the table show the maximum unroll factor used in the loop, while the columns show the column index of the 2-D array (which start from zero). The cells then show the indexing computation required to go to that column, with indices beyond the max unroll set to n/a. All multiplications and additions are done using the x86 addressing modes, while subtractions require additional registers to hold the negative values, as does L3, which holds 3*L. Note that for max unroll ≥ 9 you must consult both subtables to get all valid indices, which have been split to fit the page. . . . 103

7.2 Registers and updates needed in optimized two dimensional array of DGEMVT code shown in Figure 7.5 and benefit of this translation over general approach shown in Figure 7.4 . . . 104

B.1 Industry compilers and their versions we used in our experiment on Intel Haswell machine . . . 130

B.2 Flags used to produce vectorize and scalar code . . . 130

LIST OF FIGURES

2.1 Overview of iFKO framework 6

2.2 ATLAS's empirical search for the Level 3 BLAS 8

2.3 ATLAS+iFKO empirical search for the Level 3 BLAS 8

3.1 Example of the report on architecture 10

3.2 Example of the report of loop information for double precision AMAX kernel shown in Figure 4.1(a) using -i compiler flag 12

3.3 Format of the output of FKO's vector analyzer using -ivec2 flag 14

3.4 Example of the output of FKO's vector analyzer using -ivec2 flag for AMAX kernel 14

3.5 Register spilling report for DGEMVT with unroll and jam factor = 14 15

4.1 Kernels with multiple paths in loop: (a) Synthetic AMAX kernel (b) IAMAX kernel of level-1 BLAS 18

4.2 Code layout inside loop of IAMAX kernel: (a) CFG of paths inside loop of IAMAX (b) path-1 as fall through in code layout (c) path-2 as fall through in code layout 18

4.3 Max-Min reduction: (a) pseudocode of if-statement reduction with MAX instruction for AMAX (b) pseudocode of max variable movement with MAX instruction for IAMAX kernel 20

4.4 if conversion for single if-else sequence: (a) CFG of if-then construct (b) conversion of if-then into single block (c) CFG of if-then-else construct (d) conversion of if-then-else into single block 21

4.5 Pseudocode of if conversion for ssq loop of nrm2 before copy propagation: (a) CFG of paths inside ssq loop (d) if conversion with RC for ssq 23

4.6 Pseudocode of if conversion for AMAX and iAMAX before copy propagation: (a) if conversion with RC for AMAX (d) if conversion with RC for IAMAX . . 24

4.7 Pseudocode of IAMAX: (a) with if statement (b) if-conversion with redundant computation 25

4.8 Pseudocode of the shadow VRC vectorized SIAMAX 30

4.9 Pseudocode of the shadow VRC vectorized DIAMAX 33

4.10 Example of dscal kernel with aligned markup in HIL 36

4.11 Code generation after adding code for the peeling loop (dotted box) in vectorized code . . . 37

4.12 Example of vectorization of AXPY with loop specialization and loop peeling . . 39

5.1 Example: vectorization in the presence of unknown control flow . . . 43

5.2 Control flow graph of Figure 5.1: (a) CFG of the loop (b) Path-1 which is vectorization (c) Path-2 which is not vectorization . . . 43

5.3 SIMD vectorization using predicated execution [57] . . . 44

5.4 Proposed approach: Speculative Vectorization . . . 45

5.5 Example: analyzing the *Sum of Squares (SSQ)* kernel: (a) Scalar code (b) CFG of loop (c) Analysis . . . 50

5.6 Structure of vectorized code . . . 51

5.7 Pseudo-code for Speculatively Vectorized SSQ Loop (our compiler generates assembly) . . . 52

5.8 Transformation steps using Control Flow Graph: (a) Original CFG (b) CFG after step1 (c) CFG after step3 (d) CFG of speculative vectorized SSQ . . . 56

5.9 Speedup of tuned Speculative Vectorization over tuned unvectorized code for single precision (solid blue) and double precision (hatched red) : (a) Intel Corei2 (b) AMD Dozer . . . 61

5.10 Comparison of speedups for absolute value maximum using vectorized Max/Min Reduction (VMMR, solid blue), Speculative Vectorization (SV, diagonal hashed green) and Vectorized Redundant Computation (VRC, square-hashed orange) on Intel Corei2 for both single and double precision . . . 65

5.11 Comparison of single precision speedups on AMD Dozer for *sin* (speedups over scalar code tuned and timed for data in range [0, 2π]) using scalar code tuned and timed for data in range [-0.5, 0.5] (scal.5), Speculative Vectorization tuned and timed in range [0, 2π] (SV2pi), and range [-0.5, 0.5] (SV.5), and Vectorized Redundant Computation timed and tuned in range [0, 2π] (VRC2pi) and range [-0.5, 0.5] (VRC.5) . . . 65

6.1 Loopnests of access major martix-matrix multiplication(AMM) kernels: (a) MVEC AMM kernel with $u_m = 4$, $u_n = 4$, $u_k = 1$ (b) KVEC AMM kernel with $u_m = 4$, $u_n = 1$ and $u_k = 4$ . . . 71

6.2 Pack creation in SLP: (a) possible innermost loop (kloop) of MVEC4x4x1 kernel, (b) initial packs based on loads of pA (c) pack extension based on pA init pack (d) initial pack based on loads of pB, (e) pack extension based on pB init pack 74

6.3 SLP vectorization of kloop for MVEC4x4x1: (a) After scheduling the code based on packs (b) After emitting vector codes 75

6.4 SLP vectorization for kloop of KVEC4x1x4 : (a) kloop after renaming and accumulator expanding (c) kloop after vectorization (showing elements of vector inside box) 78

6.5 SSE implementation of VVRSUM using *veclen* = 4 and four accumulator expanded accumulators 79

6.6 AVX implementation of VVRSUM 80

6.7 Steps in loop nests vectorization 82

6.8 SLP vectorization for posttail of kloop of KVEC4x1x4 : (a) posttail after accumulator expansion (b) posttail after deleting reduction code (c) posttail after vectorizing remaining codes (d) posttail after adding vvrsum codes at the top . 83

6.9 Inconsistent vectorization: (a) variable Si is used as scalar in successor block (b) variable Si is element of different vector in successor block 86

6.10 Best-case autovectorization performance of various compilers as a percentage of the performance FKO achieves for an Intel Haswel machine 90

6.11 Autovectorizaton performance of LLVM (solid blue), ICC (right upward diagonal hashed red) and GCC (right downward diagonal hashed green) as a percentage of the performance FKO achieves for three specific *gemmµ* kernels on Intel Haswel machine. 91

6.12 Best-case performance between FKO's autovectorization (solid blue) and GCC SIMD-intrinsic (hashed red) as the percentage of performance hand tuned codes achieve in ATLAS on different machines 93

7.1 Column-major storage of a two-dimensional array with M=5, N=3 and `lda`=6 : (a) Logical storage (b) Physical memory 95

7.2 Declaration and example of pointer to two dimensional array in FKO 96

7.3 Example of two dimensional array in HIL code for DGEMVT kernel with max unroll-factor six 97

7.4 Pseudocode for the address-translation in general approach done by FKO for DGEMVT kernel shown in Figure 7.3 99

7.5 Pseudocode for the optimized address-translation done by FKO for DGEMVT kernel shown in Figure 7.3 101

ABSTRACT

iFKO (iterative Floating point Kernel Optimizer) is an open-source iterative empirical compilation framework which can be used to tune high performance computing (HPC) kernels. The goal of our research is to advance iterative empirical compilation to the degree that the performance it can achieve is comparable to that delivered by painstaking hand tuning in assembly. This will allow many HPC researchers to spend precious development time on higher level aspects of tuning such as parallelization, as well as enabling computational scientists to develop new algorithms that demand new high performance kernels. At present, algorithms that cannot use hand-tuned performance libraries tend to lose to even inferior algorithms that can.

We discuss our new autovectorization technique (speculative vectorization) which can autovectorize loops past dependent branches by speculating along frequently taken paths, even when other paths cannot be effectively vectorized. We implemented this technique in iFKO and demonstrated significant speedup for kernels that prior vectorization techniques could not optimize.

We have developed an optimization for two dimensional array indexing that is critical for allowing us to heavily unroll and jam loops without restriction from integer register pressure.

We then extended the state of the art single basic block vectorization method, SLP, to vectorize nested loops. We have also introduced optimized reductions that can retain full SIMD parallelization for the entire reduction, as well as doing loop specialization and unswitching as needed to address vector alignment issues and paths inside the loops which inhibit autovectorization. We have also implemented a critical transformation for optimal vectorization of mixed-type data. Combining all these techniques we can now fully vectorize the loopnests for our most complicated kernels, allowing us to achieve performance very close to that of hand-tuned assembly.

CHAPTER 1
INTRODUCTION

1.1 Terminology and Outline of Research

iFKO (iterative Floating point Kernel Optimizer) [62] is an iterative empirical compilation framework where the decision of what transformation set will yield the best performance is made using context sensitive timing [63] on the specific kernels and architectures being tuned for, as opposed to basing such decisions on static heuristics. The iFKO framework consists of a low level compiler and search drivers to iteratively determine the best the compiler transformations for a kernel needed to achieve high performance on a system. An overview of iFKO and its tuning framework is discussed in the Chapter 2. Our research aim is to advance this iterative and empirical compilation framework so that it is a feasible replacement for the extensive hand-tuning (often at the assembly level) common in the HPC (High Performance Computing) community. To show that compilers can achieve efficiency adequate for the HPC community, it is necessary to compare against actual HPC library routines[1] which are currently supported and tuned by the HPC community.

This research uses the BLAS (Basic Linear Algebra Subprograms) as our HPC library for validating our performance results. The BLAS is one of the most widely used high performance computing libraries in the world. It is split into three levels based roughly on kernel complexity and performance. The Level 1 BLAS [27, 33] (L1BLAS) do vector-vector operations like dot product or vector norms, and typically require only a single loop to implement. Most L1BLAS therefore do $O(N)$ computations on $O(N)$ data.

The Level 2 BLAS [17, 18] (L2BLAS) do matrix-vector operations such as matrix-vector multiply or rank-1 update, and are therefore usually implemented with at least two nested

[1] As opposed to synthetic benchmarks originally based on such libraries, which tend to overpredict compiler performance strongly due to unrealistic use-cases like statically declared operands and the possibility for whole-program analysis.

loops (one for each dimension of the matrix). They can therefore be characterized as performing $O(N^2)$ computations on $O(N^2)$ data.

The Level 3 BLAS[16] (L3BLAS) involve matrix-matrix operations such as matrix multiply or triangular (forward- and back-) solve, and are typically implemented using at least three nested loops. They perform $O(N^3)$ operations on $O(N^2)$ data.

The L1 and L2BLAS have the same order computations as data, which means that unless their operands are preloaded to the cache, they run at the speed of memory, which is orders of magnitude slower than the speed at which a modern computer can do computations. On the other hand the L3BLAS have such rich opportunities for data reuse within the memory hierarchy that they can often achieve more than 90% of the theoretical peak computational speed of the hardware.

Prior work [62] demonstrated that iFKO could be used to get hand-tuned levels of performance for all but two of the L1BLAS routines. The L1BLAS were targeted first because they are the simplest for a compiler to analyze and optimize. Note that an HPC compiler must perform each optimization almost perfectly or HPC-levels of performance cannot be achieved, and so even when there are known compiler techniques, off-the-shelf solutions are usually inadequate. The only routines that this initial research failed to adequately optimize were `IAMAX` (find the index of the maximum absolute value within a vector) and `NRM2` (safely compute the 2-norm of the vector without unnecessary floating point overflow), which iFKO could not autovectorize due to branch dependencies.

Our first major research accomplishment was to develop a novel auto-vectorization technique called *speculative vectorization* [61], which allowed us to autovectorize `NRM2` based on the SSQ (sum of squares) approach. Prior to our developing and publishing speculative vectorization, there was no known method in the literature or in commercial compilers that could achieve speedup on some types of path-based dependent loops, with SSQ being a prime example. Speculative vectorization was also able to vectorize the most frequent path

for `IAMAX`. Not only were we able to show substantial speedup using this technique, we also showed that the overhead was so low that it could plausibly be used in cases less well-suited for accurate branch prediction than SSQ or `IAMAX`. This novel transformation is discussed in detail in Chapter 5.

Our next area of research was to find a way to autovectorize all paths in `IAMAX`. We developed a technique called *shadow vectorization* to handle the mixed-type vectorization required by IAMAX, and this work is discussed in Section 4.2.1.1. Our method was a formalization for safe compilation of a hand-tuned optimization technique used in our ATLAS [64, 65, 66, 69] library. As far as we know, the first publication for this hand-tuned technique was by an Intel researcher in [8], and this appears to have been the basis for a similar transform introduced into Intel's C compiler, `icc`. At the time we developed it, our technique provided much better performance for `IDAMAX` than `icc`, but more recent `icc` versions get roughly the same performance as our implementation.

With these two fundamental extensions to the prior work, we could achieve hand-tuned levels of performance for the entire L1BLAS, and so we next studied the tuning of the L2BLAS, which feature nested loops which must be *unroll-and-jammed* [4] for decent performance. For 32-bit x86 assembly, however, we found that integer register pressure inside the innermost loop could prevent iFKO from getting good performance for the best hand-tuned unroll-and-jam factor. To fix this, we developed an optimization for 2-D array indexing that exploits the x86's powerful addressing mode, as discussed in Chapter 7. This dissertation is the first time we have published the details of this transformation, and we have so far not found any substantially similar techniques in the compilation literature.

With 2-D array addressing optimized, our main barrier to high performance for the L2 and L3BLAS was then found to be outer-loop vectorization. iFKO's existing no-hazard vectorization could successfully vectorize the innermost loop, but could not vectorize the entire loop nest, which slightly hurt L2BLAS performance, and made our autovectorized

L3BLAS uncompetitive with hand tuned codes. The state-of-the-art for general block-level autovectorization is called SLP [31] (Superword Level Parallelism). Currently, many compiler groups are exploring how best to extend SLP through arbitrary loops nests [39, 26, 58], but no standard technique has so far emerged. Widely used compilers such as Intel's `icc`, GNU's `gcc`, and the Apple-supported open source `LLVM` all had some form of outer-loop SLP support, but none of them could do an adequate job for our autotuned L3BLAS kernels. We therefore developed a more flexible extension of SLP for outer loops, as detailed in Chapter 6.

1.2 Organization of the Paper

Chapter 2 introduces the tuning framework that we used to empirically tune HPC kernels and the modification we propose in the framework to make it even better. It also describes the interface we added to integrate our empirical compilation framework with ATLAS. Chapter 3 overviews the analysis reports on kernels which our specialized compiler produces as the communication interface between the compiler and the tuning infrastructure. Chapter 4 provides a brief discussion of the transformations that we have added in our compiler to achieve higher percentage of peak performance on HPC kernels as part of this research. Chapter 5 describes our new approach, speculative vectorization, to autovectorize loops with dependent branches. Chapter 6 shows how we have extended the existing SLP vectorization technique to vectorize loopnests for ATLAS's $gemm\mu$ kernels and achieve impressive performance for such loopnests. Chapter 7 illustrates how our compiler represents two dimensional arrays and facilitates the unroll-and-jammed transformation by exploiting the powerful addressing mode of x86. Finally, Chapter 8 summarizes our contribution as well as discussing areas of future work.

CHAPTER 2
TUNING FRAMEWORK

Our ultimate goal is to provide optimized compute kernels for the HPC community that run at near-peak efficiency on increasingly powerful hardware. The traditional way to achieve high performance in HPC involves producing high performance libraries. The time critical sections of codes are first isolated. The HPC community agree on those critical sections and define the application programming interface - APIs (e.g., BLAS [27, 33, 17, 18, 16] , LAPACK [3]) for reusable performance kernels. Once those APIs are standardized, experts from different fields get together and performance tune the kernel underlying the standardized APIs. Handtuning has been used to leverage the powerful but complex hardware since traditional compilers do not achieve the required high percentage of peak. However, handtuning a kernel is very time consuming, requiring experts with knowledge of the target architecture, operation being optimized and the software layers. Moreover, handtuned codes usually are not portable from one architecture to another. These problems led to the empirically tuned library generators such as PHiPAC [9], FFTW [51, 25, 24], and ATLAS [68, 64, 65, 66, 69, 67]. The key idea behind these packages is to probe the system using empirical criteria (e.g., timing results) to evaluate the effect of each transformation and retain only those that provide measurable performance improvement on that specific system for that specific kernel. These packages have succeeded in achieving high levels of performance on a wide variety of machines, but they are limited to specific libraries. To overcome this limitation, iFKO [62, 70] (iterative Floating point Kernel Optimizer) has been designed so that these empirical tuning techniques may be applied in a compilation framework. iFKO has a backend compiler targeted to work with source generator (e.g., ATLAS's generator). It can also be used with source to source compiler (e.g., ROSE compiler [53, 46, 44], POET [73, 45], etc) and high level loop transformer (e.g., PLUTO [52, 10, 13, 5, 6, 12, 11]). In this research, we integrated iFKO in ATLAS as a case

study. In the following sections, we discuss how the iFKO and ATLAS frameworks work and how they can be interfaced together.

2.1 iFKO Framework

Figure 2.1[1] shows the overall structure of the iFKO [70, 62] compilation framework. iFKO is composed of two components: a specialized compiler (FKO), and search drivers. FKO is the specialized backend optimizing compiler for iterative and empirical use. It analyzes kernels not only to determine the legality of the transformation but also to bound the search space as an iterative compiler, performs all required transformations and generates optimized assembly codes. Two things must be supplied to iFKO by the user: the routine to be compiled (expressed in our input high level intermediate language, HIL) and a context sensitive AEOS (Automated Empirical Optimization of Software) quality timer [63] for the kernel being compiled. The HIL, similar to restricted C, is kept intentionally very simple and limited, as the initial target audience is mainly source to source generators and sophisticated hand tuners. It has extensive markup support which can be used to specify critical loops to optimize, alignment of pointers and even the safety of transformations. Note that the framework depends on externally supplied timers which are kernel-specific. In our experiments, we use ATLAS's tester and timer for this purpose.

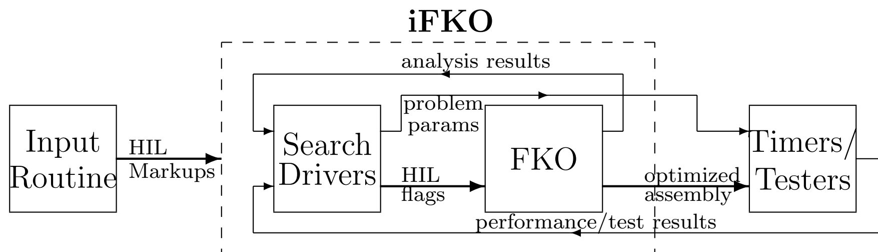


Figure 2.1: Overview of iFKO framework

[1]This figure is a modified version of a figure from the PhD dissertation [62] of iFKO by the committee chair of this research.

2.1.1 FKOC - FKO with Preprocessor

We have added a preprocessing layer to FKO which takes the lines beginning with '@' as directives from the input file with .B extension. FKO becomes FKOC with this layer. FKOC actually uses ATLAS's *extract* program internally to preprocess the input code and in order to generate an output file with .b extension, which FKO recognizes as a input language. FKOC can emulate many capabilities of C's preprocessor (e.g., macro substitution). Since FKOC internally uses *extract*, it has some scripting abilities (e.g., looping structures, integer arithmetic, etc) which are not present in C preprocessor. ATLAS extensively uses CPP macros in its generated codes. We adapted ATLAS's current L3BLAS generators to produce .B files for use with FKOC, so that we can directly compare FKO and other compilers for performance tuning. FKOC then calls FKO to compile the codes.

2.2 ATLAS Framework

Figure 2.2 outlines the search of matmul kernels in ATLAS and Figure 2.3 shows how iFKO can be interfaced with ATLAS to tune the same L3BLAS kernels[2]. ATLAS uses multiple layer of searches to tune L3BLAS kernels. The master search probes the machine for system specific information (e.g., L1 cache size, FPU unit, pipeline, etc). The master search then calls the source-generator search which uses heuristics to probe the optimization space allowed by the source generator and returns the parameter settings (e.g., blocking and unrolling factors, etc) of the best cases. The master search then calls the multiple implementation search which times all the hand written implementations and returns the best one. The best performing kernel (found using the empirical results provided by the AEOS-quality timer) among the results of the generator and multiple implementation searches is then taken as a system specific kernel (see [62] for more details).

[2]Figure 2.2 is collected and Figure 2.3 is modified from the PhD dissertation [62] of iFKO by the committee chair of this research.

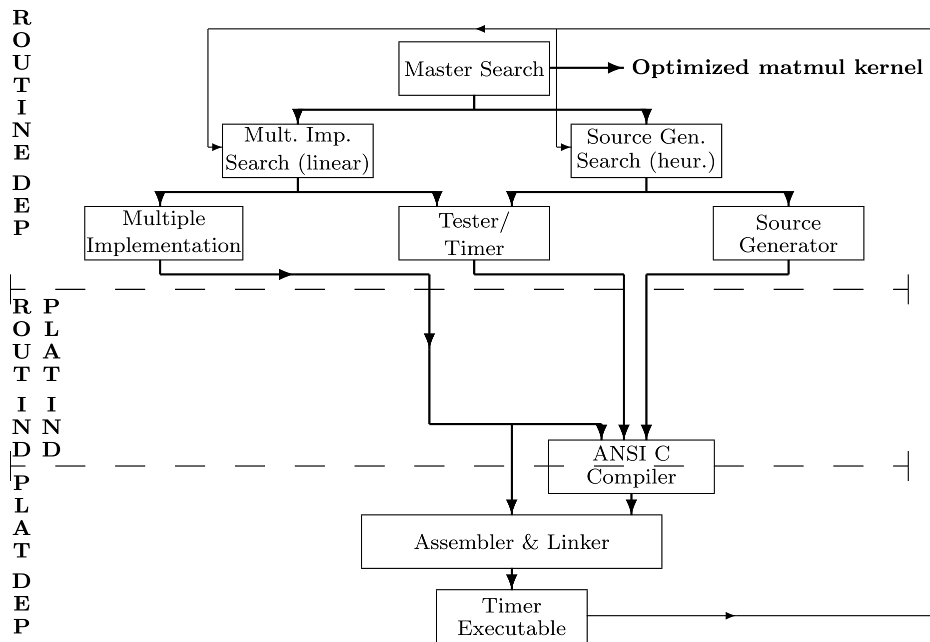


Figure 2.2: ATLAS's empirical search for the Level 3 BLAS

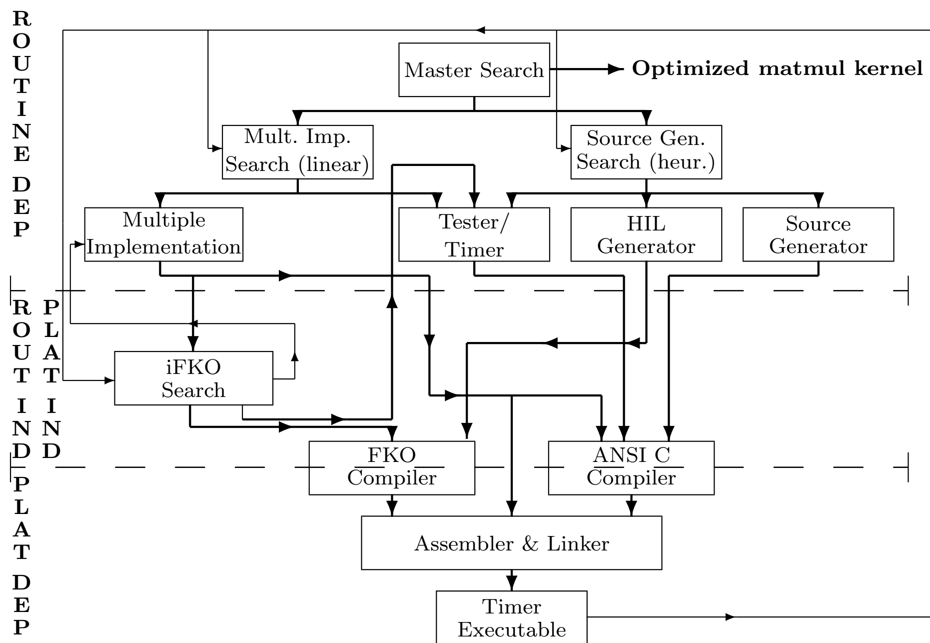


Figure 2.3: ATLAS+iFKO empirical search for the Level 3 BLAS

As shown in Figure 2.2, ATLAS defaults to using an ANSI C compiler to compile all kernels and non-kernel codes (e.g., timer, tester, etc). The user can configure separate compilers to compile different type of kernels (e.g., matmul, non-matmul, etc). ATLAS uses similar tuning process (less complex than the tuning of L3BLAS) to tune other level of BLAS kernels.

2.3 Interfacing ATLAS and iFKO

The components of iFKO (compiler and search) are independent. Therefore, the framework of iFKO can be used as a whole (FKO+search) or as standalone compiler (FKO or FKOC), where the iteration is left to the ATLAS tuning framework. Figure 2.3 outlines the different interfaces of iFKO with ATLAS. As shown in the figure, iFKO can be used with ATLAS's pre-existing multiple implementation support. ATLAS then treats iFKO as another kernel compiler taking as input the kernel expressed in FKO's HIL. The empirical tuning of iFKO is independent and therefore potentially complementary to ATLAS's empirical search. For example, ATLAS can tune the block factor, outer loop unroll and jam of a matmul kernel by the source generator search while iFKO does the tuning of innermost loop unrolling, scalar expansion and/or prefetching in this setup.

However, FKO (FKOC) can also be used as a standalone kernel compiler and leave the empirical tuning to completely to ATLAS's searches. We have added a HIL generator in ATLAS as shown in the Figure 2.3. This HIL generator is similar to the ATLAS's source generator but it generates scalar kernels in FKO's HIL, including exploiting its extensive markup capabilities. We use FKOC (FKO) to perform all the transformations and optimizations for the generated kernels which we have implemented as part of this research (discussed in the Chapter 3 to 7). We can use similar interfaces with other source to source generator and/or empirical tuning framework. In future work, we will investigate such integrations.

CHAPTER 3
ANALYSIS IN FKO TO AID IN SEARCH

FKO performs various analysis on kernels not only to determine the legality of transforms but also as an interface between the compiler and the empirical tuning search, which can use these analysis to bound the search space. We keep these analysis reports human readable and independent from the iFKO's native search so that FKO can easily serve as a backend compiler for other source-to-source automated tuning frameworks, various custom search drivers, and hand tuners interested in extracting maximal performance without writing directly in assembly. In this chapter, we will overview some of the more important of FKO's current analysis reports.

3.1 Architecture Analysis

FKO provides report on architecture when $-iarch$ flag is used. This report includes pipeline, register, cache, SIMD vector and instruction specific information about the system. This information can be used to bound the search in tuning. For example, we can limit the search to find the best unroll and jam factor for L2BLAS kernels (e.g., DGEMVT) by the number of registers since we need to avoid the spilling registers inside the innermost loop to get competitive performance. Figure 3.1 shows an example of FKO's architecture report on one of our machines. Line 1 of the Figure 3.1 shows the pipeline information. Here the value zero means that information about the pipeline is unknown to FKO. This system has six types

```
1  PIPELINES=0
2  REGTYPES=6
3      NUMREGS: i=15 f=16 d=16 vi=16 vf=16 vd=16
4      ALIASGROUPS=1
5          ALIASED: f d vi vf vd
6  NCACHES=3
7      LINESIZES: 64 64 64
8  VECTYPES=3
9      VECLEN: i=8 f=8 d=4
10 EXTENDEDINST=3
11     MAXINST: f d vi vf vd
12     MININST: f d vi vf vd
13     CONDMOV: i f d vi vf vd
```

Figure 3.1: Example of the report on architecture

of registers as shown in line 2. Line 3 shows the count for each of these six register types with 15 integer (the dedicated stack pointer not included in this count), 16 single precision floats, 16 double precision floats, and 16 of each vector registers. Note that all the vector register types and floating point registers are aliased (line 4 to 5). This system has three layers of caches and the cache line of each layer of caches is 64 bytes (line 7). It supports three types for vector instructions (line 8) and the length of vector for each type is specified in line 9. Line 10 shows it has three extended instructions, line 11 to 13 show the supported types for each of these extended instructions. FKO has a configuration file where all of these information are specified. As of now, most of these values are filled in when we port FKO to a machine, but in the future many of them will be empirically discovered automatically.

3.2 Optloop Analysis

Optloop analysis is one of the most important reports FKO provides to search. The optloop is loop specified by the user as the main source of performance; the optloop is defined by a special syntax in the input language of FKO. The compiler generates optloop information when the $-i$ flag is passed. This report consists of information of paths, vectorization methods, moving pointers and scalars. Figure 3.2 shows an example of such report on the optloop for AMAX kernel. Line 1 specifies whether there is any optloop in the kernel. If there is none, there will only be one line in the report and the value of the OPTLOOP will be zero. Line 2 shows the number of paths inside the optloop. Here it is two. Line 3 shows the vectorizability of each paths. One of the paths is vectorizable in this kernel. Line 4 shows the methods to remove all the non-loop branches and hence, to reduce all paths into a single path. We can use two methods to remove the branches for this kernel: max reduction and if conversion with redundant computation (discussed in Chapter 4). We then have information about the if-statement (if-then and if-then-else constructs) as shown in line 5 to line 8. This kernel has only one if-statement and this if-statement can be removed using the same two methods we mentioned before. Line 9 shows the applicable vectorization methods for this

```
1  OPTLOOP=1
2     NUMPATHS=2
3        VECTORIZABLE: 1 0
4        EliminatedAllBranches: MaxMin RedComp
5        NUMIFS=1
6           MaxEliminatedIfs=1
7           MinEliminatedIfs=0
8           RedCompEliminatedIfs=1
9     VECTORIZATION: LoopLvl SpecVec
10    Moving 1D Pointers: 1
11       'X': type=d uses=2 sets=1 lds=1 sts=0 prefetch=1
12    Scalars Used in Loop: 2
13       'amax': type=d uses=1 sets=1 ReduceExpandable=1
14       'x': type=d uses=3 sets=2 ReduceExpandable=0
```

Figure 3.2: Example of the report of loop information for double precision AMAX kernel shown in Figure 4.1(a) using -i compiler flag

kernel. We can apply loop vectorization (after reducing all the paths into one) and speculative vectorization for this kernel. This report also provides information about the moving pointers (which are incremented by constant inside loop) in line 10 to 11 and scalars inside optloop in line 12 to 14). This simple kernel has one moving pointer inside the loop. Line 11 provides information about the type of the pointer along with the number of static use/def and memory loads/stores (using this pointer) inside optloop. The memory access using this pointer is also prefetchable, meaning it is a candidate to use software prefetch instruction tuned by the search. Line 12 to Line 14 provides information about the scalar variables. This kernel has two double precision floating point scalars. It provides use/def information inside the optloop for them as well. One of the scalar variables (`amax`) is scalar expandable, meaning we can apply the scalar expansion optimization (for unrolling or vectorizing the kernel) to this variable. The search driver can find the best combination of loop unroll factor and scalar expansion of this variable for this kernel during the tuning step.

3.3 Vectorization Analysis

FKO supports three different auto-vectorization methods for the optloop: no-hazard loop vectorization (NHV), speculative vectorization (SV) and superword level vectorization (SLP). A complete list of compiler flags in FKO related to vectorization is shown in Table 3.1. We can apply any vectorization on the optloop directly by throwing the flags. The vectorization is automatically extended towards the outer loops by SLP vectorization if the loop-nests

Table 3.1: Vectorization related flags of FKO

Flag	Description
-LNZV	Apply no-hazard loop vectorization on optloop, may apply SLP on rest of the loopnest if possible
-SV (ipath) (nvlens)	Apply speculative vectorization on path i of optloop, speculated iteration = nvlens
-SLP, -SLP (id)	Apply SLP on optloop, apply special sequence for init pack of SLP
-ivec, -ivec2	provide report of different vectorization methods
-vec	apply best vectorization method estimated by the analysis (SV not considered due to path dependence)
-ibvec	print the id of the estimated best vectorization method
-vecapproach (id)	apply a specific vectorization method

satisfy a special pattern[1] (see Chapter 6 for details). To apply no-hazard vectorization, we can use $-LNZV$ flag. Note that if the analysis of the validity of the vectorization fails, FKO throws an error (with explanatory message). The $-SV$ flag is used to apply speculative vectorization. Note the additional arguments of the flag. The *ipath* is the path number (provided by the optloop analysis) which SV should speculatively vectorize. The path number zero means the default fall through path. The *nvlens* is the argument for the larger bet unrolling (see Chapter 5 for details), where 1 means the count of the speculated iteration is equal to the vector-length elements (*vlen*) and 2 means the speculative iteration is $2 \times vlen$ and so on. To apply SLP on the optloop, we use $-SLP$ flag. It has an optional argument to specify the ordering of the seed packs of the SLP (see Chapter 6 for details). The value of *id* can be found by applying the $-ivec2$ flag. The $-ivec2$ flag is used to get the vectorization report in details while $-ivec$ provides the summery of the report (first two lines of the full report). Figure 3.3 shows the format of the vectorization report and the Figure 3.4 shows an example of the vectorization report on AMAX kernel given by $-ivec2$ flag. This kernel can be vectorized by speculative vectorization and the first path is vectorizable as shown in line 1 in Figure 3.4. We can vectorize the kernel by two other ways (other than speculation)

[1]Extending vectorization beyond optloop from speculative vectorization is not supported yet.

```
1      SPECVECBYPATH : <list 0/1 for each path>, <0> not applicable
2      VECTORIZATION : <nways>; <list of id of vecmethods sorted by best to worst>
3         OPTLOOP : [LoopLvl=10/11,12 Slp=1001,1002,1002]
4         VecRankLvls : <vlvl>
5            lvl1: list of vecid
6            lvl12: list of vecid
7            ...
```

Figure 3.3: Format of the output of FKO's vector analyzer using -ivec2 flag

```
1   SPECVECBYPATH:   1 0
2   VECTORIZATION: 2; 12, 13,
3      OPTLOOP: LoopLvl=12,13, SLP=0,
4         LoopLvl:   MaxMin=12 RedComp=13
5      VecRankLvls: 1
6            1: 12, 13,
```

Figure 3.4: Example of the output of FKO's vector analyzer using -ivec2 flag for AMAX kernel

and the *id* number of them are 12 and 13 (line 2). The next two lines show the meaning of those ids. Both of them eventually indicate no hazard vectorization but after applying different path reduction methods. No-hazard vectorization is applied after using max/min reduction transformation in case of $id = 12$ and if conversion with redundant computation for $id = 13$. SLP vectorization is not applicable ($SLP = 0$) for this kernel. The vectorization report also provides information about the rank (estimated) of the methods. This rank is calculated based on the level of nested loops the method can vectorize. The innermost loop is the most important, then the outer loop of the innermost loop and so on. Since AMAX has one loop, the rank of both the methods are same. Within the same rank, it is sorted by the priority of the methods. For example, max/min reduction is generally superior to if conversion after redundant computation and therefore, $id = 12$ is estimated as the best method to vectorize the kernel. Using $-vec$ flag, we can automatically apply this estimated best method of vectorization. The *id* of the estimated best method is the first entry in rank 1 (12 in line 6). We can print the *id* of the estimated best vectorization method (excepting speculation vectorization[2]) by using $-ibvec$ flag as well. We can even apply vectorization by

[2]We skip the speculative vectorization in our ranking system since the profitability of this method strongly depends on the paths taken at runtime.

```
1  RSPILLS=2                          /* number of scopes to show */
2     OPTLOOP: i=2 vd=2               /* spilling in optloop */
3     GLOBAL: i=13 d=2 vd=9           /* spilling in global scope, meaning the whole routine*/
```

Figure 3.5: Register spilling report for DGEMVT with unroll and jam factor = 14

using the any of the vectorization *id* when we use $-vecapproach$ flag. Therefore, we have great flexibility for the user/search to try different methods of vectorization by FKO.

3.4 Register Spilling Analysis

FKO provides the register spill information of the output assembly for any kernel when the flag $-ilrs$ is used. The proper assignment of registers in the innermost loop is very crucial to achieve high performance for any kernel. Some transformations may increase register pressure inside the loop (e.g., unroll and jam, scheduling of instructions, etc). The tuning framework (search) can also use this information to safely bound an optimization search. Figure 3.5 shows an example of the live-range spilling for the DGEMVT kernel with unroll and jam factor 14. Note the first line of the report. It specifies the number of scopes. We are supporting two scopes in our current implementation: the optloop and the routine (global). Line 2 and 3 provide the spill count for the optloop and the entire routine.

CHAPTER 4

TRANSFORMATIONS ADDED IN FKO TO OPTIMIZE BLAS KERNELS

FKO is specially designed to optimize HPC kernels under an empirical tuning framework. Unlike in general purpose compilation, it is better to be able to fully tune a narrow range of kernels than to partially tune any kernel: a only moderately tuned kernel is not useful for HPC. In order to best use our R&D time, we only add a transformation (and its attendant analysis), when we have a real-world kernel that requires it for HPC-competitive performance. In the below list, we overview the motivation and transformations that we have undertaken as part of this thesis work:

- We have added path based transformations to optimize kernels with conditional branches inside loops. The original FKO [62] failed to achieve good performance on IAMAX and NRM2 (SSQ variant) of the L1BLAS kernels because branches inside the loop prevented autovectorization. This was a significant example of the fact that branches not only affect performance adversely when misprediction occurs, but also inhibit other compiler optimizations which may provide critical speedups. Therefore, to overcome the adverse effect of branches, we implemented several path based transformations in FKO, as discussed in Section 4.1.

- Autovectorization is one of the most important compiler optimizations since SIMD units are ubiquitous in modern microprocessors. We have not only updated the traditional loop vectorization in FKO to support our shadow VRC vectorization (discussed in Section 4.2.1.1), but also implemented two additional autovectorization techniques. One of the methods, Speculative Vectorization (SV) [61], is a novel way to autovectorize loops with conditional branches (discussed in Section 4.2.2 and Chapter 5); the other is an extension of well known Superword Level Parallelization (SLP) vectorization (discussed in Section 4.2.3 and Chapter 6). SV autovectorization enables FKO

to achieve excellent performance for NRM2 (not effectively vectorizable by any other known method) and IAMAX kernels when tuned with the search driver. SLP, on the other hand, helps FKO to achieve high efficiency for kernels with nested loops (e.g., ATLAS's *gemm*μ).

- We have added additional x86-specific optimizations in FKO as well. We exploit the powerful addressing modes of the x86 to optimize the memory addressing of two dimensional arrays in FKO. Our representation of 2D arrays minimizes the register usage and computations to manage the memory addressing for such arrays. This special representation and optimization of the 2D array (discussed in Section 4.4 and Chapter 7) helps FKO to obtain good performance for L2BLAS kernels on this architecture.

4.1 Path Based Optimization

We have implemented several path based transformations in FKO. Those transformations help FKO to achieve competitive performance to hand-tuned code for those kernels which have loop-carried dependent branches inside loops. Some path based transformations provide significant performance boost (e.g., path reduction transformations), while some of them facilitate other transformations (e.g., frequent path coalescing is used before speculative vectorization). FKO by default explores and analyzes all paths inside innermost loop in its path based optimization. The user can also provide a threshold to limit its search if exploring all paths is too costly. Figure 4.1 shows two kernels with two paths inside the loop. The AMAX kernel in Figure 4.1(a) is a synthetic kernel which finds the absolute value maximum from an array and IAMAX in Figure 4.1(b) is one of the L1BLAS kernels which is used to find out the index of the absolute value maximum from the array. We will use these examples to describe our path based optimizations[1]. In following sections, we will describe some of those path based transformations in brief.

[1]FKO's optloop is essentially a do-while loop. However, since for-loop is more common in C codes, we use the syntax of the for-loop in most our pseudocodes examples where it does not impede understanding.

```
1  amax = 0.0;
2
3  for (i=0; i < N; i++)
4  {
5      ax = X[i];
6      ax = fabs(ax);
7      if (ax > amax)
8      {
9          amax = ax;
10     }
11
12 }
```

(a)

```
1  amax = 0.0;
2  imax = 0;
3  for (i=0; i < N; i++)
4  {
5      ax = X[i];
6      ax = fabs(ax);
7      if (ax > amax)
8      {
9          amax = ax;
10         imax = i;
11     }
12 }
```

(b)

Figure 4.1: Kernels with multiple paths in loop: (a) Synthetic AMAX kernel (b) IAMAX kernel of level-1 BLAS

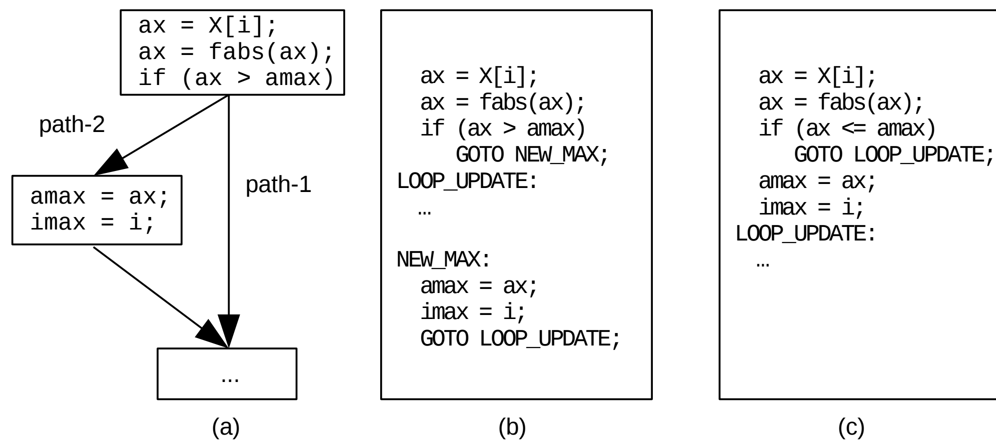


Figure 4.2: Code layout inside loop of IAMAX kernel: (a) CFG of paths inside loop of IAMAX (b) path-1 as fall through in code layout (c) path-2 as fall through in code layout

4.1.1 Frequent Path Coalescing

FKO uses one explicit branch target in its intermediate language (LIL). In general, a taken branch must be correctly predicted to avoid large performance penalties. Therefore, choosing the frequent path in the loop as the fall through often yields better performance, since the fall-through frequent path would not cause a pipeline flush even in the complete absence of branch prediction. The loop analyzer of FKO enumerates all paths inside loop. FKO can make a specified path fall through in the code by (possibly) rearranging its CFG and inverting the conditions of branches. Figure 4.2 shows paths of the IAMAX kernel inside the loop and the code layout when each of the paths is made fall through. The branch inside the loop creates two paths for this kernel (as shown in Figure 4.2(a)). In Figure 4.2(b), *path*1

is by default fall through and therefore the instructions in *path*1 are contiguous in memory (increasing spatial locality for frequent path). In Figure 4.2(c), *path*2 is made fall through. We implement this transformation by swapping the conditional and unconditional successor in the CFG after inverting the condition of the branch. The tuning framework of FKO can empirically tune the code by making the most important path (as guided by the timing) as fall through using this transformation.

4.1.2 Path Reduction

Branches with loop carried dependences create multiple paths inside loop. Multiple paths may inhibit many compiler optimizations including vectorization. Therefore, reducing multiple paths into one by eliminating branches not only simplifies the complexity of the CFG of code, but also facilitates other code optimizations. In FKO, we have several methods to remove the branches from the loop. We will discuss our path reduction techniques in Sections 4.1.2.1 and 4.1.2.2.

4.1.2.1 Max-Min Reduction

We found several kernels in ATLAS (e.g., IAMAX, IRKxAMAX) where a variable is used in a loop to find the maximum value in an array. These kernels perform different computations but they have common code patterns where the max value is computed. We implemented an analyzer which can recognize the pattern of such max/min variable by analyzing the variable, the conditionals and the loop. We have also implemented a transformation which can push the max/min variable out of the if-statement using MAX/MIN instruction (if architecture supports such instructions). Figure 4.3(a) shows how we can use this transformation in AMAX from Figure 4.1(a) and Figure 4.3(b) shows transformed code from the IAMAX in Figure 4.1(b). In Figures 4.3(a) and (b), `_MAX` represents the max instruction provided by the hardware. In case of the AMAX kernel, we can remove the if-statement by using max instruction as shown in Figure 4.3(a). After this transformation, the IAMAX kernel still has a branch as shown in Figure 4.1(b). However, we can also use if-conversion with redundant

```
1  amax = 0.0;
2
3  for (i=0; i < N; i++)
4  {
5
6      ax = X[i];
7      ax = fabs(ax);
8      amax = _MAX(ax, amax);
9
10 }
```

(a)

```
1  amax = 0.0;
2  imax = 0;
3  for (i=0; i < N; i++)
4  {
5      ax = X[i];
6      ax = fabs(ax);
7      if (ax > amax)
8         imax = i;
9      amax = _MAX(ax, amax);
10 }
```

(b)

Figure 4.3: Max-Min reduction: (a) pseudocode of if-statement reduction with MAX instruction for AMAX (b) pseudocode of max variable movement with MAX instruction for IAMAX kernel

computation (which we describe next, in Section 4.1.2.2) to reduce the paths and remove the branch of IAMAX using select (blend) operation.

4.1.2.2 If Conversion with Redundant Computation (RC)

The main idea of if conversion [2] is to convert control dependencies into data dependencies and thus eliminate conditional branches. This can be almost always succeed if the hardware supports predicating arbitrary instruction. However only a few architectures support predicating all instructions. On the x86, we have only a small number of instructions which can be used as effective predication. For example, there are special SIMD *compare* instructions (e.g., vcmpxx) which can store the result of a conditional evaluation in mask register. The results of two different former computational paths can then be selected into their final destination register from their temporary computation registers using the results stored in the mask register by using the select (AKA: blend) instruction. We utilize these *compare* and *blend* instructions to remove branches. To delete a branch, we save the result of *compare* statement, redundantly compute both the paths, and select the correct value from those redundant computations using the *blend* (select) operation (as in [7, 57, 49]). We perform the following two steps in order to eliminate the innermost if-then and if-then-else constructs:

1. **Find if-then and if-then-else constructs in the CFG:** As a first step, we need to find the if-then and if-then-else constructs in the CFG: our technique is similar to [49],

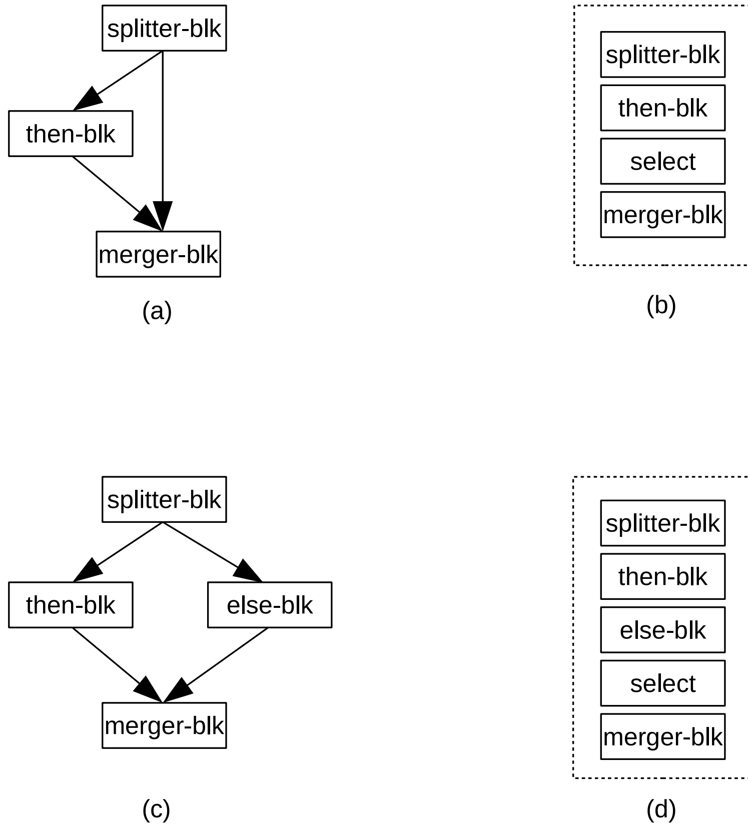


Figure 4.4: if conversion for single if-else sequence: (a) CFG of if-then construct (b) conversion of if-then into single block (c) CFG of if-then-else construct (d) conversion of if-then-else into single block

as implemented in the SUIF [54, 72, 71] compiler. Figure 4.4(a) shows the CFG of a single if-then construct. Note that the *splitter* block of the if-then construct has two successors: the *then* block (the body of if-statement) and the *merger* block (where the two paths meet). The two successors of if-then-else construct (shown in Figure 4.4(c)) however are *then* block and *else* block. The successor of both of these blocks is the *merger* block. We use these observations to recognize the innermost if-then and if-then-else constructs. Note that these successor-predecessor relationships may only be true for the innermost construct in the nested if-else statements. However, we can recognize the immediate outer construct (in nested case) of the innermost one using the same observations but after converting the innermost construct into a single block. We describe how we convert those constructs into single blocks in next step.

2. **Convert if-then and if-then-else construct into a single block by eliminating the branch and using the select operation:** This step converts the if-then and if-then-else constructs in the CFG into a single block by placing the statements of *splitter*, *then* and *else* (if exists) blocks consecutively and adding the select statements before the *merger* block as shown in Figure 4.4(b) and Figure 4.4(d). Before converting the CFG, we need to eliminate the branch in the *splitter* block and rename the variables (which are defined) in the *then* and *else* blocks. To eliminate the branch, we first convert the *compare* instruction which effects condition code (EFLAGS in x86) with the *compare* instruction which saves the result in a register based on the conditional jump in FKO's intermediate representation (LIL). For example, *compare* instruction `FCMP` followed by the conditional jump `JEQ` are replaced by single compare instruction `FCMPWEQ` which saves the result in a register (`fmask`) in FKO's IR (LIL) as shown here:

$$\texttt{FCMPWEQ fmask, freg0, freg1} \begin{cases} \texttt{FCMP fcc0, freg0, freg1} \\ \\ \texttt{JEQ pcreg, fcc0, label} \end{cases}$$

We then rename those variables which are set/defined in the *then* and *else* blocks. We rename their successive usages inside the blocks as well. We then use select(blend) instruction to select the correct value from those two versions of (renamed) variables using the previously generated mask.

Iterative algorithm to eliminate all if-else constructs in loop : Our iterative algorithm to eliminate all branches in loop (except the loop-branch) works as follows. We first delete the back edge in CFG for loop and find the innermost if-then/if-then-else construct. We then apply our RC transformation (discussed in the Step 2) to reduce the construct into a single block. After reconstructing the CFG, we repeat the process until there are no if-then/if-then-else constructs left to transform.

Figure 4.5 shows an example of how we reduce the paths to single path inside the *ssq* loop using our if-conversion algorithm. Figure 4.5(a) shows the if-then-else construction inside the

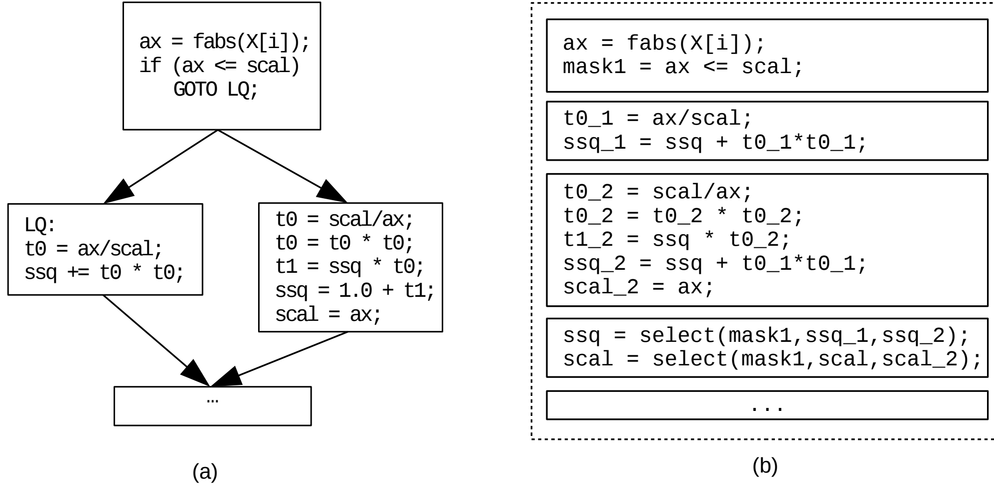


Figure 4.5: Pseudocode of if conversion for ssq loop of nrm2 before copy propagation: (a) CFG of paths inside ssq loop (d) if conversion with RC for ssq

loop. We first convert the *compare* statement by saving the result of the *compare* in `mask1` and removed the branch in *splitter* block. In *then* block, we rename the variables: `t0` and `ssq` whereas in *else* block, we rename the variables: `t0`, `t1`, `ssq` and `scal`. We place these modified statements of the *splitter*, *then* and *else* blocks consecutively in a single block. We then add select(blend) statements to choose the correct value from the pair of the renamed variables based on the `mask1` as shown in Figure 4.5(b). Note that we need to add the select statement only for `ssq` and `scal` since they are live-in to the *merge* block whereas we do not need select statements for other variables since they are local/private. Note the select statement for `scal`: since we do not have any new definition of `scal` in the *then* block, we used `scal` (original) and `scal_2` (defined in *else* block) in the select statement.

Figure 4.6 shows the pseudocode after if-conversion with RC for AMAX and IAMAX kernels originally shown in Figure 4.1. Figure 4.6(a) shows the if-conversion of the AMAX kernel. Note that we use select (blend) instruction here whereas in Figure 4.3(a) we use `max` instruction. Both of these approaches are valid for AMAX, but only if-conversion with RC can remove all non-loop branches completely for IAMAX. Figure 4.6(b) shows how we can remove all non-loop branches by using the select operations (more details in Section 4.1.2.3).

```
1  amax = 0.0;
2
3  for (i=0; i < N; i++)
4  {
5
6     ax = X[i];
7     ax = fabs(ax);
8     mask1 = (ax > amax);
9     amax1 = ax;
10    amax = select(mask1, amax1, amax);
11
12 }
```

(a)

```
1  amax = 0.0;
2  imax = 0;
3  for (i=0; i < N; i++)
4  {
5     ax = X[i];
6     ax = fabs(ax);
7     mask1 = (ax > amax);
8     amax1 = ax;
9     imax1 = i;
10    amax = select(mask1, amax1, amax);
11    imax = select(mask1, imax1, imax);
12 }
```

(b)

Figure 4.6: Pseudocode of if conversion for AMAX and iAMAX before copy propagation: (a) if conversion with RC for AMAX (d) if conversion with RC for IAMAX

In our current implementation, we allow only floating point *compare* to trigger if-conversion with redundant computation. Moreover, redundant computation may not always be valid. Consider an if-then-else construct where one path accesses a valid memory address whereas other path accesses an invalid address and in normal execution, only the path with valid memory address would execute. However, the redundant computation will try to execute both paths and thus generate exceptions. Finding all the exceptional cases is impossible for an HPC *library*, since it involves whole program pointer analysis [28, 59, 47]. We depend on the user of FKO to utilize this transformation only when redundant computation is safe. We have markup which specifies that redundant computation is not safe for a given loop, in which case FKO will not consider it for that loop body.

4.1.2.3 Redundant Computation for Mixed-type Data Using Derived Masks

FKO is targeted towards floating point kernels, and so presently we only support applying the path reducing transformations (redundant computation, max/min conversion) for ifs whose comparisons are floating point (adding support for path reductions with integral comparisons would be straightforward, and will be done if an important kernel requiring it is brought to our attention).

However, it is quite possible for a floating point *comparison* `if` to contain computations of types that differ from the parent comparison. If such variables are live when leaving the if or

```
1  amax = 0.0;
2  imax = 0;
3  i=0;
4  do
5  {
6      ax = X[i];
7      ax = fabs(ax);
8      if (ax > amax)
9      {
10         amax = ax;
11         imax = i;
12     }
13     i++;
14 }while(i < N);
```

(a)

```
1  amax = 0.0;
2  imax = 0;
3  i=0;
4  do
5  {
6      ax = X[i];
7      ax = fabs(ax);
8      fmsk = (ax > amax);
9      imsk = _FP2INTBITS(fmsk);
10     amax = select(fmsk, ax, amax);
11     imax1 = i;
12     imax = select(imsk, imax1, imax);
13     i++;
14 }while(i < N);
```

(b)

Figure 4.7: Pseudocode of IAMAX: (a) with if statement (b) if-conversion with redundant computation

else block, they must be selected based on the original differing-type comparison. On many architectures, selections with types different from the comparison can require additional work.

Presently, FKO only supports the mixed type live-out redundant computation that our kernels have required. The only case presently supported is mixing a particular floating point precision with non-pointer integral operations (other mixed type computations are supported if they are not live out of the if/else block(s), and thus do not need to be selected).

To simplify prior discussions, we have so far elided the fact that the IAMAX kernel first introduced in Figure 4.1(b) is an example of mixed type data, and cannot be handled quite as simply as previously shown.

To see the problem, examine Figure 4.7(a) (which presents the same kernel as Figure 4.1(b), but in do-while loop format). Line 8 shows an `if` with a floating point comparison, which means the comparison supports path reduction via redundant computation. However, on line 11 this floating point comparison causes the update of `imax`, which is a 32-bit integer that is the return value of the function (which means this set will be live past the if body). Line 10's `amax` is of the same type as the comparison (the precision could be a 32-bit float or 64-bit double depending on declaration, but in IAMAX all floating point variables have the same precision).

The complication comes on line 12 of Figure 4.7(b) where we would like to select either the the value of `i` or pre-existing `imax` value using `select()`. For the x86, FKO does all floating point computations (even scalar operations) using the vector unit(s), while almost all integer operations utilize the ALU. The ALU cannot directly read the vector registers, and since line 9's `fmsk` is assigned to vector register, we must therefore convert the floating point comparison result into something accessible by the ALU.

Pseudocode for this is shown in line 9; using the synthetic operation `_FP2ICC`, which moves the bits stored in `fmsk` to the ALU-accessible register `imsk`, which can then be used as the predicate for the integer select operation on line 12. In practice, line 9 is done using an instruction like `MOVMASKPD` or `MOVMASKPS`, while line 12 requires using `BT` (bit test) to set the integer condition codes, with the actual selection done by conditional move (which reads the integer condition codes set by by `BT`).

Note that FKO can be trivially extended to support other mixed types by converting between masks as necessary, but this has a cost as we have seen here. As explained above, in addition to doing the redundant computation, we would add an additional 2 instructions to convert the floating point predicate mask to an integer. It is sometimes possible to avoid predicate type conversion costs, and one technique that can sometimes be used to avoid these costs is called *shadowing*.

The idea of shadowing can be applied anytime the comparison type is at least as big as the type being used in a dependent select, and the functional units handling each type can directly access the same register sets. In this case, we can store the bits making up the select computations within a region where the comparison type would normally be (i.e. in the "shadow" cast by the comparison type), as long as we select the correct bits to read, and make sure they are not type converted. We can then use a floating point select to select an integer value, for instance, avoiding the extra costs required to convert the predicate mask

between types. FKO presently applies comparison shadowing only when autovectorizing mixed integral and floating point operations, as discussed in Section 4.2.1.1.

4.2 SIMD Vectorization

SIMD vector units are ubiquitous in modern microprocessors. Therefore, their effective utilization is critical to attaining high performance for the HPC kernels. The original FKO [62] supported no-hazard loop autovectorization to vectorize the innermost loop. We have not only extended this vectorization to vectorize our path-reduced codes, but also implemented two new autovectorization methods. Therefore, FKO supports three different methods for the autovectorization of the innermost loop: no-hazard loop vectorization (NHV), speculative vectorization (SV) and superword level vectorization (SLP). We have also implemented a special method to autovectorize the nested loops. If the innermost loop of the nested loops is vectorized by NHV or SLP, we can then extend the vectorization to the outer loops by using our SLP vectorization. We will discuss each of the autovectorization methods in following sections.

4.2.1 No-Hazard Loop Vectorization

No hazard loop vectorization(NHV) is the extended version of the SIMD vectorization supported by the original FKO [62]. It is basically a loop level vectorization which vectorizes the innermost loop when there are no non-loop conditional branches. It performs an analysis to recognize the patterns of the scalars and to ensure the validity of the transformation. We have extended this vectorization to recognize the special statements (e.g., *select*, *max*) added by our path reduction transformations. In addition to this, we have added support to recognize and vectorize induction variables. Moreover, we have added the "shadowing" technique (described in Section 4.1.2.3) to efficiently vectorize if-converted codes with some mixed-type data (e.g., IAMAX). We will describe this technique in following section.

4.2.1.1 Shadow Vectorization After Redundant Computation

If conversion with redundant computation (RC) introduces special *compare* and *select* statements which our no-hazard vectorization converts into *vector-compare* and *vector-select*. The *vector-compare* statement compares two vectors and saves the results in a *vector-mask*. This *vector-mask* contains the boolean results computed by the logical comparisons between each elements of the two vectors. Consider a SIMD vector of single precision floating point in AVX2[2]. The AVX2 SIMD unit operates on 256 bits at once, and so operates on eight 32-bit single precision floating point values at a time. We therefore have eight results in a *vector-mask* after using *vector-compare* for this type. Consider now the vectorization of single precision IAMAX kernel after shadowing (a similar technique is shown in [8] for SSE unit using assembly code). We start with a transformed kernel after applying the path reduction by RC as shown in Figure 4.7(b). The API of IAMAX uses 32-bit integers. On the x8664, FKO normally promotes 32-bit integers into 64 bits during the function prologue so that they may be freely used in addressing (x8664 expects all components of an addressing mode to be 64 bits). Recognizing that the API only calls for 32-bit values however, this promotion will not be done on the *vectorized* integers derived from the index, so that our integer and float have the same length. Note that if the API dictated 64-bit integers, we would need two SIMD integral vectors to hold the same number of vector elements as the eight 32-bit floats, and so we would have to apply mask conversion as in Figure 4.7(b), rather than shadowing with the same mask, as we can do using 32-bit integers.

In Figure 4.7(b), we save the result of the *compare* statement ($ax > amax$) in the `fmsk` (line 10). The `fmsk` is converted to `imsk` to select correct value from the two integers (`imax1` and `imax` in line 12). Let us consider 32 bit integer for this example. Both the single precision float and the integer vector have eight elements since x86 uses same SIMD unit for all the

[2]Our current implementation works only for AVX2 and in x8664. We will extend to other SIMD vectorization schemes in future work.

types. We therefore can use the same *vector mask* (produce by the *vector compare*) to choose correct integer (32 bit) elements using the *vector select* statement (unlike the `select()` using `imsk` in scalar code). Vectorizing the *select* statement for `amax` is trivial, but `imax` requires further analysis and special handling in the vector loop, vector-prologue and vector-epilogue. This section will discuss this process in detail. Figure 4.8 shows the pseudocode after applying the shadow vectorization for the single precision IAMAX kernel. The `vimax1` vector simulates the index `i` of the vector loop, which is implicitly unrolled by 8. Note the initialization of the `vimax1` and `vvl` on lines 9 and 10, respectively. In each vector iteration, `vimax1` keeps track of the eight values of `i` that the scalar would traverse when unrolled by the vector length of 8. Therefore, each elements of `vimax1` is incremented by vector-length ($veclen = 8$) using `vvl` in line 19 in each vector iteration. We then use the mask `vmask1` produced by the vector *compare* in line 18 to update `vamax` so that each of its eight floats will contain the maximum of its original value and the values held in `vamax` (see line 20).

Since this is IAMAX, we need to compute the maximum index the max value came from, and since a 32-bit integer and 32-float are of the same size, and the AVX2 vector unit uses the same registers for floats and integers, we can store the index containing the mask in the shadow of the max value that it results from. We can therefore use shadow vectorization on the index computation as well, and utilize the exact same `vmask1` to compute the index corresponding to the max value, as seen on line 21 of Figure 4.8.

After executing all iterations of the vectorized loop, `vamax` contains 8 different maximum values found at strided locations, while `vimax` contains the indices corresponding to those partial maxes. The problem is that the maximum value is not necessarily unique. In this case, note that IAMAX *requires* us to return the minimum index of the non-unique (tying/equal) maximum values found (as dictated by the condition on line 8 and loop condition on line 14 of Figure 4.7(b)).

```
1  #define maxInt 0xEFFFFFFF
2  amax = 0.0;
3  imax = 0;
4  i = 0;
5
6  // vector prologue
7  vamax = [amax, amax, amax, amax, amax, amax, amax, amax];
8  vimax = [imax, imax, imax, imax, imax, imax, imax, imax];
9  vimax1 = [i-8, i-7, i-6, i-5, i-4, i-3, i-2, i-1];
10 vvl = [8, 8, 8, 8, 8, 8, 8, 8];
11
12 // vector loop
13 do
14 {
15    // loop body
16    vax = X[i:i+7];
17    vax = vfabs(vax);
18    vmask1 = (vax > vamax);
19    vimax1 = vimax1 + vvl;
20    vamax = vselect(vmask1, vax, vamax);
21    vimax = vselect(vmask1, vimax1, vimax);
22    // loop update
23    i += 8;
24 } while(i < N);
25
26 // vector epilogue
27
28 // step1: Reduce amax from vamax: amax = HMAX(vamax)
29 vamax0 = vamax;                     /* save 8 partial max before reduction */
30 Va = _VSHUF(vamax, 0x7654FEDC);  /* upper half to lower half */
31 vamax = _VMAX(Va, vamax);
32 Va = _VSHUF(vamax, 0x765432BA);  /* 3rd and 4th to 1st and 2nd position */
33 vamax = _VMAX(Va, vamax);
34 Va = _VSHUF(vamax, 0x76543219);  /* 2nd to 1st position */
35 vamax = _VMAX(Va, vamax);
36 amax = _VHSEL(vamax, 0);          /* set amax with the 1st element in vector */
37
38 // step2: generate vmask2
39 Vb = [amax, amax, amax, amax, amax, amax, amax, amax];
40 vmask2 = (vamax0 == Vb);           /* mask true if given elt ties for amaxval */
41
42 // step 3: select appropriate elements of vimax using vmask2
43 vmaxInt = [maxInt, maxInt, maxInt, maxInt, maxInt, maxInt, maxInt, maxInt];
44 vimax = vselect(vmask2, vimax, vmaxInt);
45
46 // step4: Reduce imax from vimax: imax = HMIN(vimax)
47 Vi = _VSHUF(vimax, 0x7654FEDC);  /* upper half to lower half */
48 vimax = _VMIN(Vi, vimax);
49 Vi = _VSHUF(vimax, 0x765432BA);  /* 3rd and 4th to 1st and 2nd position */
50 vimax = _VMIN(Vi, vimax);
51 Vi = _VSHUF(vimax, 0x76543219);  /* 2nd to 1st position */
52 vimax = _VMIN(Vi, vimax);
53 imax = _VHSEL(vimax, 0);          /* set imax with the 1st element in vector */
```

Figure 4.8: Pseudocode of the shadow VRC vectorized SIAMAX

Two reductions for which FKO had pre-existing support are *horizontal maximum* and *minimum*, meaning finding the max/min value stored amongst the vector length elements of a vector register. We use this stock reduction to compute the (possibly non-unique) maximum value from the eight partial max values in lines 30-36 of Figure 4.8. This reduction involves recursive halving: its first vector max (line 30) only produces useful values in half the vector, and half the remaining parallelism is lost at each of the $(log_2(veclen) - 1)$ vector maximums, until we have the scalar result we wanted in the low element of the `vamax` vector register, and we can then move that value into a scalar register, as shown in line 36. Note that this reduction is done outside the loop, and so is a lower order cost than the loop vectorization it enables.

We have now computed the scalar maximum absolute value from its vector representation inside the loop, and now we must do the same for its corresponding index, presently stored in `vimax`. If every single element of `vamax` had the (equal) maximum value, we could just reduce `viamax` using the horizontal (integer) vector minimum in like fashion, but of course this is extremely unlikely. Instead, each element of `viamax` will contain a integer $i, 0 \leq i \leq maxInt$, where `maxInt` is the maximum storable positive integer.

What we are now going to do is replace every index within `vimax` that does not contain the maximum found absolute value with `maxInt`. If this is done, we can compute the correct index to return by doing a horizontal minimum. The proof for this is quite straightforward: if any maximum value was found at a number less than `maxInt`, then the `maxInt` replacement values will be discarded during the horizontal minimum, and we will return the minimum tying value as required. If the maximum value is uniquely found at `maxInt`, then the horizontal minimum will be `maxInt`, which again is the correct scalar return value as defined by IAMAX.

On line 39 we load the recently computed (possibly non-unique) scalar maximum to all elements of the vector `Vb`. We now compare this all-max vector with the copy of the original partial max vector `vamax0` (save of loop's `vamax` on line 29, with comparison with broadcast

maximum on line 40). Line 43 produces the vector vmaxInt with the broadcast value `maxInt`, which we then use to replace any non-maxval entries using vector select. We now find the return value to IAMAX using our standard $log_2(veclen)$ horizontal vector minimum, as seen on lines 47-52, with line 53 setting the integer return value from the reduced vector of indices[3].

Figure 4.9 shows pseudocode for our vectorized double precision IAMAX kernel. Since doubles are 64 bit in size, we keep FKO's standard promotion of the API's 32-bit integer to internal 64 bit, so that our integers and doubles fit in the same space. The AVX2 SIMD vector unit now operates on four double precision floating point values or four 64 bit integer values at a time. We therefore can use the same mask for the shadowing again. So, this case looks a lot like the last, except the halved vector loop trip count: Line 7 shows the initialization of the absolute value maximum vector (`vamax`) with the 4 double elements. Line 8 shows the initialization of the index vector `vimax`. Finally, `vv1`, used to increment the vector index count, has all values set to 4 (line 10) to indicate we process four doubles at once with one vector loop iteration. The loop (lines 13-24) works exactly same as before. The reduction steps in vector epilogue are also similar. The process of the reduction of `vamax` into `amax` is same but requires one less step (line 30 to 34) than before since the $veclen = 4$. We update the non-maxval indices of `vimax` with `maxInt` as before. However, the reduction of `vimax` to `imax` is different (line 45 to 51) from the single precision IAMAX (see line 29 to 36 in Figure 4.8). Since AVX2 does not support MAX/MIN vector operation for 64 bit integer values, we use *select* operations with the mask (`vmask3`) generated from the comparison `Vi > vimax` in each of $log_2(veclen)$ steps (line 47 and 50). Therefore, each `_VMIN` instruction for integer vector is converted into a *vector comparison* followed by a *select* operation (line 46 to 47 and 49 to 50) and we get the final result which is set to `imax` in line 51.

[3]FKO internally sign-extends this 32 bit value into 64 bit to store it back into the x8664's 64-bit general purpose register.

```
1  #define maxInt 0xEFFFFFFFFFFFFFFF
2  amax = 0.0;
3  imax = 0;
4  i = 0;
5
6  // vector prologue
7  vamax = [amax, amax, amax, amax];
8  vimax = [imax, imax, imax, imax];
9  vimax1 = [i-4, i-3, i-2, i-1];
10 vvl = [4, 4, 4, 4];
11
12 // vector loop
13 do
14 {
15    // loop body
16    vax = X[i:i+3];
17    vax = vfabs(vax);
18    vmask1 = (vax > vamax);
19    vimax1 = vimax1 + vvl;
20    vamax = vselect(vmask1, vax, vamax);
21    vimax = vselect(vmask1, vimax1, vimax);
22    // loop update
23    i += 4;
24 } while(i < N);
25
26 // vector epilogue
27
28 // step1: Reduce amax from vamax: amax = HMAX(vamax)
29 vamax0 = vamax;
30 Va = _VSHUF(vamax, 0x3276);   /* upper half to lower half */
31 vamax = _VMAX(Va, vamax);
32 Va = _VSHUF(vamax, 0x3215);   /* 2nd to 1st position */
33 vamax = _VMAX(Va, vamax);
34 amax = _VHSEL(vamax, 0);      /* 1st element */
35
36 // step2: generate vmask2
37 Vb = [amax, amax, amax, amax];
38 vmask2 = (vamax0 == Vb);
39
40 // step 3: select appropriate elements of vimax using vmask2
41 vmaxInt = [maxInt, maxInt, maxInt, maxInt];
42 vimax = vselect(vmask2, vimax, vmaxInt);
43
44 // step4: Reduce imax from vimax: imax = HMIN(vimax) implemented using select
45 Vi = _VSHUF(vimax, 0x3276);   /* upper half to lower half */
46 vmask3 = (Vi > vimax);
47 vimax = vselect(vmask3, Vi, vimax);
48 Vi = _VSHUF(vimax, 0x3215);   /* 2nd to 1st position */
49 vmask3 = (Vi > vimax);
50 vimax = vselect(vmask3, Vi, vimax);
51 imax = _VHSEL(vimax, 0);     /* 1st element */
```

Figure 4.9: Pseudocode of the shadow VRC vectorized DIAMAX

4.2.2 Speculative Vectorization

We implement a new approach, speculative vectorization [61], which speculates past dependent branches to aggressively vectorize computational paths that are expected to be taken frequently at runtime, while simply restarting the calculation using scalar instructions when speculation fails. We have integrated our technique in iFKO's tuning framework to employ empirical tuning to select paths for speculation. iFKO has achieved up to 6.8X speedup for single precision and 3.4X for double precision kernels using AVX in our studied kernels, while increasing performance for some operations (e.g., *ssq* loop of $nrm2$) that could not be sped up by any prior vectorization technique. Chapter 5 describes this technique in detail.

4.2.3 Superword Level parallelization (SLP) Vectorization

Superword Level Parallelism (SLP)[31] is the state of the art method for auto-vectorizing the straight line code in any basic block. The main idea of SLP is to exploit ILP by scheduling isomorphic statements (statements which contain the same operations in the same order) to pack them together into vector operations. We have extended this single basic block SLP vectorization to vectorize nesting loops. We have applied this method on ATLAS's $gemm\mu$ microkernels and achieved significant speedup over the autovectorizations of other industry compilers. We will describe this technique in Chapter 6.

4.2.4 Vector Intrinsic/Language Support

We have added *vector* types in the input language (HIL) of FKO. We have also added some preliminary vector operations to provide the programmer a way to express vectorized code directly using the high level intermediate language (HIL). Table 4.1 shows the vector operations FKO supported so far. It is still in its elementary phase. We will expand this support as needed in future.

4.3 SIMD Alignment Issues

The original FKO always assumed to be 128-bit aligned to vector loads and stores which only supported SSE (128 bit) unit. ATLAS's framework was exploited to guarantee this

alignment for the studied kernels. We extended vectorizations for AVX (256 bit) unit and implemented several methods to handle alignment for both the units. We will discuss how we handle alignment in loop vectorizations (e.g., NHV and SV) in Section 4.3.1 and in SLP vectorization in Section 4.3.2.

4.3.1 Alignment in Loop Vectorization

In our extension of FKO, we added support to handle arbitrary alignment of memory address in the vector loop in x86. However, we do not want to add any extra overhead for any kernel which is known to be aligned to the size of the SIMD vector. Therefore, we introduced a special markup (annotation), *ALIGNED*, for the innermost loop to specify the known alignment of any address at the starting of the loop iterations. Figure 4.10 shows how we can utilize this markup to specify the alignment of a pointer inside a loop. Consider the AVX SIMD unit for this example. We specify the X pointer as 32 byte aligned in line 11.

Table 4.1: Vector operations supported in FKO

Operation	Example	Description
Declaration	VDOUBLE (VLEN) :: Vx, Vy;	VLEN is integer constant which represents the element count in vector
Memory Load	Vx = X[0];	load VLEN elements specified in declaration
Memory Broadcast	Vx = _VBROADCAST(X[0]);	loads single element of X, but broadcast it to vetcor
Memory Store	X[0] = Vx;	stores the vector on stating address of X
Arithmetic Ops	Vx = Vy + Vz; Vx = Vy - Vz; Vx = Vy * Vz; Vx = Vy / Vz; Vx += Vy * Vz;	all the operands must be vector
Vector Initialization	Vx = {x,x,x,x}; Vx={0,0,0.0,0.0,x};	

Vector Reduction	x = _VHADD(Vx);
	x = _VHSUB ;
	x = _VHMAX(Vx);
	x = _VHMIN(Vx);
	x = _VHSEL(Vx,const_pos);
		for scalar element select (HSEL), const_pos is a int constant from 0 to (vlen-1).

```
1  ROUTINE ATL_USCAL
2      PARAMS :: N, alpha, X, incX;
3      INT :: N, incX;
4      DOUBLE :: alpha;
5      DOUBLE_PTR :: X;
6  ROUT_LOCALS
7      INT :: i;
8      DOUBLE :: ax;
9  ROUT_BEGIN
10     LOOP i = 0, N
11     ALIGNED(32) :: X;    // X is known to be 32 byte aligned
12     LOOP_BODY
13         ax = X[0];
14         ax = ax * alpha;
15         X[0] = ax;
16         X += 1;
17     LOOP_END
18 ROUT_END
```

Figure 4.10: Example of dscal kernel with aligned markup in HIL

Note that it is a loop markup that means the memory address X points to is at least 32 byte aligned on the first iteration of the loop. We can safely use aligned vector-load to load data from X after vectorization (in AVX). However, if the alignment of X in this kernel is not known (e.g., no markup), we peel the loop to force X to be aligned before entering the vector loop. In the vector loop, we can then use aligned loads and stores of X. We will describe how we generate codes with loop peeling in following section and we will discuss a more general case with more than one array in Section 4.3.1.2.

4.3.1.1 Loop Peeling to Handle Alignment

In peeling for alignment, the iterations of the loop are peeled by appropriate number of scalar iterations until the relevant pointer is aligned to the required alignment before entering the vector loop. With the inclusion of the peel loop[4], we now have three separate loops in our vectorized code: peel loop, vector loop and cleanup loop. Since both peel loop and cleanup are scalar loops, we can implement them with a single scalar loop. However, in order to keep the implementation simple, we keep a separate loop for the peeling. Figure 4.11 shows a flow chart of the generated code after introducing the peel loop in vectorization. Consider the

[4]loop peeling can be implemented and optimized without any scalar loop (e.g., we do not need any loop at all when the vector length is 2), but to handle the general case we implement loop peeling with a scalar loop, which we call the “peel loop”.

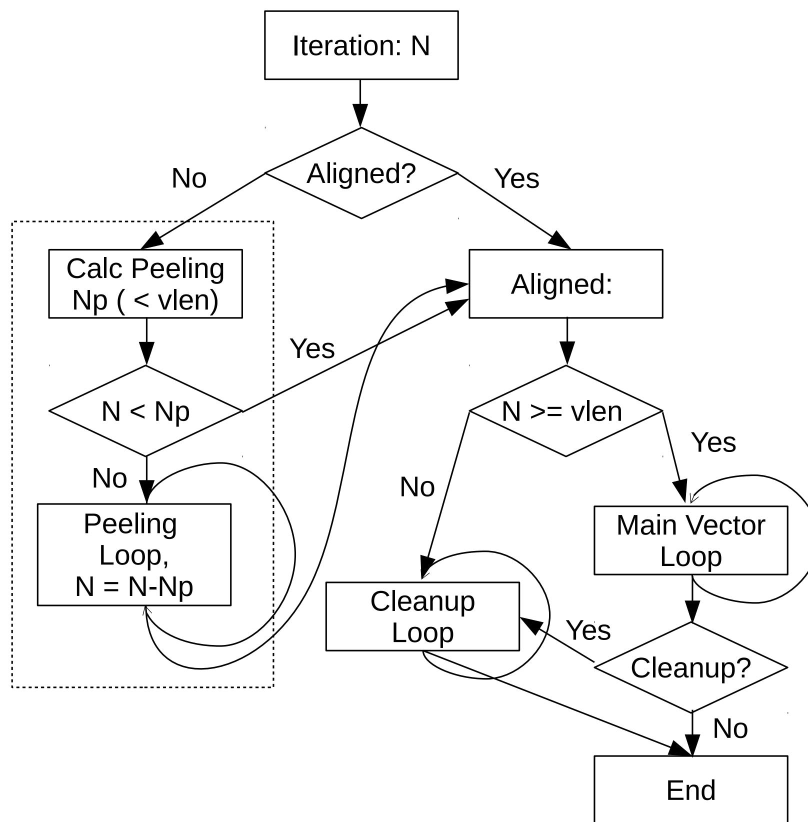


Figure 4.11: Code generation after adding code for the peeling loop (dotted box) in vectorized code

example of Figure 4.10 but without markup. The loop iteration count in this example is N. We first test the alignment of X. If it is already aligned, we can execute our vectorized loop without peeling. However, if it is not aligned, we jump to our new generated code segment as shown inside the dotted box in Figure 4.11. We then compute the scalar loop iteration needed, Np, to make X aligned. If Np is greater than the original N, we don't need the peeling; the program will eventually execute the cleanup loop since we do not have enough iterations to execute the vector loop. We will execute the peeling loop to make X aligned otherwise. We then jump to the aligned section of the code with remaining iteration count N as $N - Np$. We now can execute the previous aligned vector loop. If all the pointers are mutually aligned in case of multiple pointers, we can still apply this loop peeling to make all of them aligned to the required bytes. FKO supports a loop markup (MUTUALLY_ALIGNED) to specify the mutual alignment of the pointers in FKO.

4.3.1.2 Loop Specialization

If two or more pointers are mutually misaligned, we cannot make all of them aligned with a peel loop. The general solution to this problem is to force the alignment of one of the pointers via loop peeling as we described before and generate the vector-loop with the assumption that the given pointer is aligned and the rest of the pointers are not aligned. We analyze the loop body to find the most accessed pointer (read and write) in loop as the candidate of this forced aligning. However, when we do not have any knowledge of the mutual alignment of pointers, we generate a duplicated vector loop (as a special loop) where we assume all of them are aligned (the best case scenario). Figure 4.12 shows an example of the loop specialization to handle such alignment. In this example, we show loop specialization for $AXPY$ kernel to handle the alignment. Since Y is the most accessed pointer inside the loop, we make it our candidate pointer for the loop peeling. We introduce a markup (FORCE_ALIGN) in FKO so that user can suggest the candidate pointer as well. After executing the peeling loop, we check whether the X has also become aligned. If it is true, we will then execute the vector

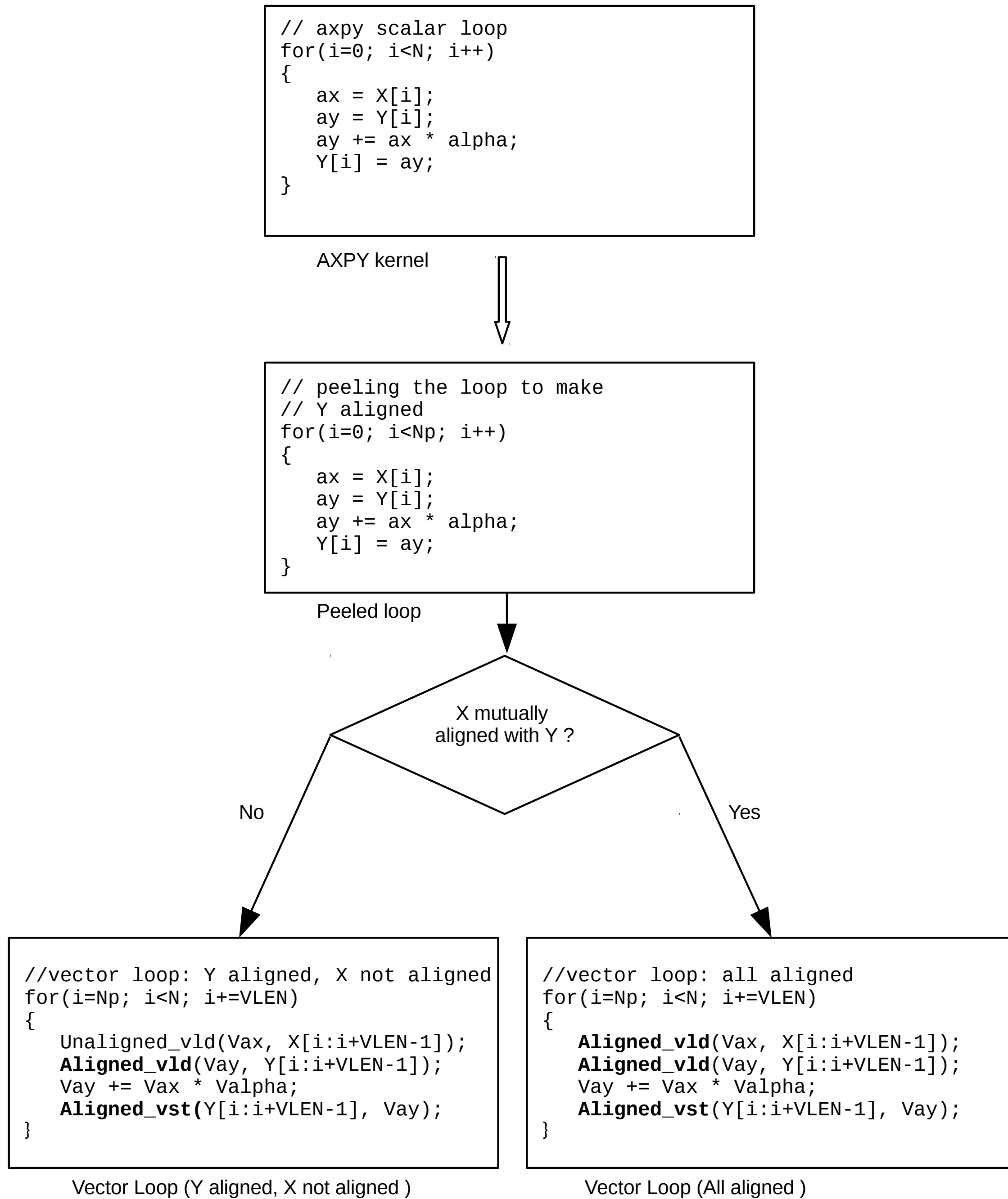


Figure 4.12: Example of vectorization of AXPY with loop specialization and loop peeling

loop with all aligned loads and stores (our best case). Otherwise, we will execute the vector loop assuming only Y aligned and X unaligned.

Table 4.2 lists all the markups and examples of their usage to handle the alignment in FKO. If all pointers are aligned, we do not need any special code to handle alignment. If all pointers are mutually aligned, we only need the peeling loop to make them aligned. If we do not have any markup for the alignment and the FORCE_ALIGN is used to suggest the candidate pointer, we make that pointer aligned by the loop peeling and then apply loop specialization. If there is no markup in loop, we find the candidate pointer by analyzing the code in loop and then apply the loop specialization. Multiple markups can be also used at the same time to precisely specify a scenario.

4.3.2 Alignment in SLP

SLP vectorization is normally applied on straight line code of a single block. In FKO, SLP can also be applied to vectorize nested loops. We do not consider the markup of innermost loop for the outer loops. Therefore, by default FKO assumes all pointers in outer loops are unaligned. We introduced routine markup to specify the alignment of the pointers at the routine level so that we can consider them aligned in the outer loops. A thorough analysis to detect the alignment of the pointers based on the given hint and/or innerloop alignment will be considered in future.

Table 4.2: Examples of the usage of loop markups to handle alignment in FKO. Consider a loop with X, Y, Z arrays and the length of SIMD vector vl bytes

Cases	Description	Loop Peeling	Loop Specialization
ALIGNED(vl)::X,Y,Z;	all aligned, X%vl = Y%vl = Z%vl = 0	no need	no need
MUTUALLY_ALIGNED(vl):: X,Y,Z;	all mutually aligned, X%vl = Y%vl = Z%vl	Yes, X	no need
FORCE_ALIGN::X	Make X aligned	Yes, X	Yes
No Markup	No knowledge of alignment	Yes, most accessed	Yes

4.4 Architecture Specific Optimization

We have implemented x86 specific optimizations on our compiler. Optimization on the two dimensional array is worth mentioning here. FKO supports two dimensional column major arrays where the elements within a column are consecutive and the elements within a row are strided. FKO can exploit the rich addressing modes of the x86 and minimize the registers required to hold column pointers and update operations inside the unrolled and jammed loop. This optimization is key for unrolled and jammed level-2 and block-major level-3 BLAS kernels. We will describe this optimization in Chapter 7.

4.5 Summary and Conclusions

This chapter presents the transformations we have added to the backend compiler FKO of the open source empirical compilation framework iFKO. Besides adding various path based transformations, we have implemented two new vectorization techniques. We will describe them with results in separate chapters later. Thanks to these techniques, all the BLAS kernels in ATLAS can be effectively autovectorized with the performance close to the handtuned codes. We have implemented several strategies to handle SIMD alignment proposed in the original dissertation of iFKO [62]. In addition to this, we have also implemented architecture specific optimizations. Considering all these optimizations, iFKO can competitively be used in ATLAS (in place of ATLAS's intrinsic generator).

CHAPTER 5
SPECULATIVE VECTORIZATION

This chapter is previously published at PACT 2013 [61][1]. With SIMD vector units becoming ubiquitous in modern microprocessors (e.g., x86 SSE/AVX, ARM NEON, POWERPC AltiVec/VMX, among others), their effective utilization is critical to attaining a high level of performance for scientific applications. Most compilers, e.g., GNU gcc and Intel icc, can automatically vectorize instruction sequences when safe [55, 23, 38]. However, when instructions are embedded inside conditional branches, their vectorization is often inhibited due to the presence of unknown control flow. Existing research has exploited predicated execution of vectorized instructions [57, 56] to support SIMD vectorization of such instructions. However, without special hardware support, these techniques need to evaluate all the branches of a control flow before using special instructions to combine results from different branches, resulting in a significant amount of replicated computation whose results are never used. Figure 5.1 illustrates this problem with a loop nest that includes partially vectorizable statements inside control flow branches. In particular, the statement $s1$ can be fully vectorized, $s2$ can be vectorized with predicated execution, and $s3$ cannot be vectorized due to loop-carried dependences. Figure 5.2 shows the control flow graph of these statements, where both s1 and s2 can be safely vectorized if *Path-1* is taken at every vectorized iteration of the surrounding loop. Figure 5.3 shows the result of vectorization using the predicated execution approach of Shin et al. [57]. Here s1, p1, and s2 are all vectorized, with the result of the vectorized p1 (vpT) serving as a mask in selecting the valid results of s2. Then, the predicate vector vpT is unpacked and used to selectively evaluate the four unrolled instances of s3. Note that s2, now translated into two vectorized instructions, is always evaluated irrespective of

[1]This chapter previously appeared as [Majedul Haque Sujon, R. Clint Whaley, and Qing Yi. Vectorization past dependent branches through speculation, published by The Institute of Electrical and Electronics Engineers (IEEE)]. See the letter in Appendix C.

```
1  for (i=1; i<=1024; i++)
2  {
3  s1:    a = A[i] * scal; /* vectorizable */
4  p1:if (a <= MaxVal)
5  s2:       B[i] = A[i];   /* vectorizable */
6      else
7  s3:       B[i] = B[i-1]; /* not vectorizable */
8  }
```

Figure 5.1: Example: vectorization in the presence of unknown control flow

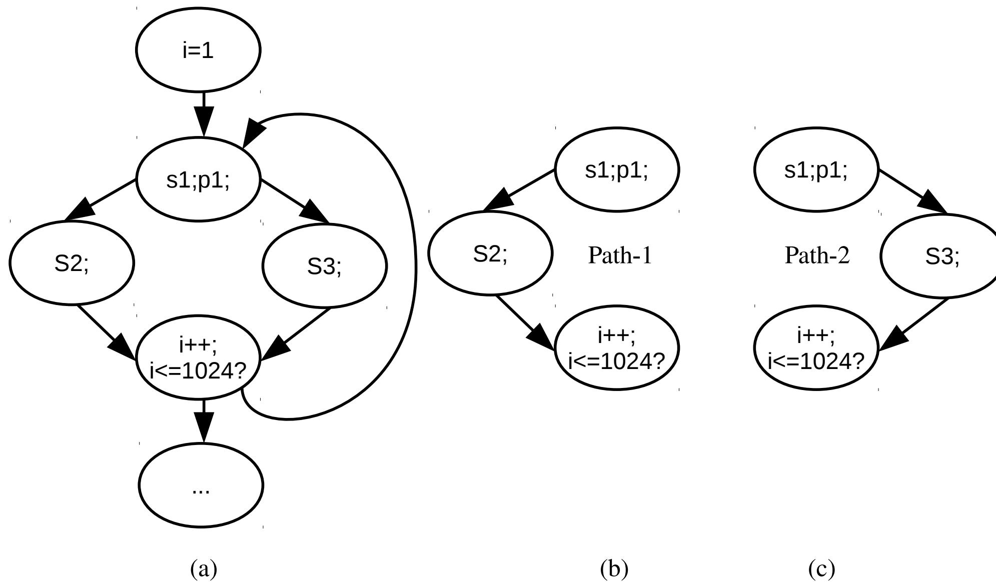


Figure 5.2: Control flow graph of Figure 5.1: (a) CFG of the loop (b) Path-1 which is vectorization (c) Path-2 which is not vectorization

```
1   for (i=1; i<=1024; i+=4)
2   {
3  s1: Va[0:3] = A[i:i+3] * [scal,scal,scal,scal];
4  p1: vcomp = Va[0:3] <= [MaxVal,MaxVal,MaxVal,MaxVal];
5  s2: vpT, vpF = vpset(vcomp);
6      B[i:i+3] = select(B[i:i+3],A[i:i+3],vpT);
7  s3: /* scalar part */
8      [PF1,PF2,PF3,PF4] = UNPACK(vpF);
9      if(PF1) B[i] = B[i-1];
10     if(PF2) B[i+1] = B[i];
11     if(PF3) B[i+2] = B[i+1];
12     if(PF4) B[i+3] = B[i+2];
13  }
```

Figure 5.3: SIMD vectorization using predicated execution [57]

the output of the predicates. Further, the unpacking of the predicate vector vpT could result in extra pipeline stall cycles within the CPU. In this paper, we present a new approach, which speculates past dependent branches to enable aggressive vectorization of paths that are evaluated frequently at runtime. As illustrated in Figure 5.4, where the path composed of statements $s1$ and $s2$ is selected and speculatively parallelized, our approach checks the correctness of the speculation at a very early stage, and if the speculation fails, the alternative scalar iterations (s3 in Figure 5.4) are evaluated instead. In addition to allowing the vectorization of routines that cannot be vectorized by existing techniques, our experimental results show that this speculative vectorization approach can outperform existing techniques when the control flow branches are strongly directional; that is, the vectorized path is frequently taken at runtime (e.g., kernels such as MAX/MIN). However, in situations where control flow paths are unpredictable (i.e., a random branch could be taken at any iteration), overly high misspeculation rate could result in our approach performing worse than the original code or code vectorized via predication. To ameliorate this limitation, we use an iterative compilation framework [62] to experiment with different path speculations so that the technique is applied only when beneficial for representative inputs. We have implemented our speculative vectorization technique within iFKO [62], an iterative optimizing compiler that focuses on backend optimizations for computation-intensive floating point kernels which uses empirical tuning to automatically select the best performing transformations, and have used

```
1  for (i=1; i<=1024; i+=4)
2  {
3  s1:Va[0:3] = A[i:i+3] * [scal,scal,scal,scal];
4  p1:if (Va[0:3] <= [MaxVal,MaxVal,MaxVal,MaxVal])
5  s2:    B[i:i+3] = A[i:i+3];
6     else /* Scalar Restart */
7     {
8  s3:     for (j=0; j<4; j++)
9          {
10            a = A[i+j] *scal;
11            if (a <= MaxVal)
12               B[i+j] = A[i+j];
13            else
14               B[i+j] = B[i-1+j];
15         }
16     }
17 }
```

Figure 5.4: Proposed approach: Speculative Vectorization

iFKO to perform SIMD vectorization for 9 floating point benchmarks with single and double precision variants. Our results show that up to 6.8X speedup for single precision and up to 3.4X speedup for double precision can be attained for these benchmarks in AVX through our speculative vectorization optimization. Our contributions include the following:

- We present a new approach for speculatively vectorizing loops past dependent branches and a path-based vectorization analysis algorithm for automatically identifying opportunities for applying speculative vectorization.

- We have integrated our technique within an iterative compiler and used empirical tuning techniques to automatically select the most profitable path to vectorize.

- We demonstrate the effectiveness of our techniques using a large number of floating point kernels, including some inherently scalar code, e.g., the sum of square computation for nrm2 in the BLAS library [29], which could not be vectorized efficiently using existing techniques.

The remainder of the chapter is organized as follows. Section 5.1 describes our algorithm for speculative vectorization. Section 5.2 summarizes our integration of the algorithm within the iFKO iterative optimizing compiler framework. Section 5.3 describes our experimental

methodology and the results obtained using speculative vectorization. Section 5.4 presents related work, and Section 5.5 presents our conclusions.

5.1 Description of Speculative Vectorization

Our research aims to support aggressive SIMD vectorization of important loops even when their bodies contain complex control-flow and when the entire computation cannot be fully parallelized. Our solution effectively combines two classes of existing techniques, SIMD vectorization and path-based speculation, that have been highly successful in modern compilers.

To speculatively vectorize a loop, we first find all possible paths through the loop body. Analysis is then performed to determine which of these paths can be safely vectorized, and the set of safely vectorizable paths are returned to the search engine of iFKO, an iterative compilation framework for backend optimization. The search driver of iFKO invokes its optimizing compiler to experimentally vectorize iterations of statements along each path, measures the performance of the differently vectorized code, and finally selects the most profitable path to be vectorized for the original application. The following subsections present both the analysis and transformation steps in detail. The overall iFKO iterative compilation framework is then outlined in Section 5.2.1.

5.1.1 Safety Analysis

Algorithm 5.1 outlines the main steps of our safety analysis algorithm, which takes a single input loop and returns a set of paths that are safe targets for speculative vectorization.

In more detail the three steps of Algorithm 5.1 are:

1. *is_loop_form_vectorizable*: In step 1, we determine whether the input loop is in a form suitable for vectorization. In particular, we require that the loop must be regular (i.e., can be easily translated to a Fortran Do-style loop) and countable [36], where the number of iterations of the loop is known before entering the loop body, and all loop iterations can be counted using an integer index variable. If the input loop fails to satisfy this condition, it is considered unsafe to vectorize.

Algorithm 5.1: speculative vectorization analysis

```
funct is_loop_vectorizable(loop)
  (1)if (!is_loop_form_vectorizable(loop))
       then return(empty);  fi
  (2)paths =  select_paths_to_speculate(loop);
  (3)foreach p ∈ paths
       do
       if (!is_path_vectorizable(p, loop))
         then paths = remove(p,paths);  fi
  od
  return(vectorizable_paths)
end
```

2. *select_paths_to_speculate*: In step 2, we determine which paths are candidates for speculative vectorization. Since the cost of finding all paths through a loop body could grow exponentially as the number of branches increases, our compiler takes an optional command-line argument that sets the maximum number of paths to consider for vectorization, and any remaining paths will not be considered once the threshold is exceeded. This threshold is implemented to ensure our optimization is never overwhelmed by overly complex control flow, which is not expected to happen often in practice. In particular, iFKO, the compiler infrastructure where we implemented our optimization, targets floating point kernels, which typically have fairly modest control flow complexity, and having too many paths to analyze almost never becomes a concern. Our compiler analyzed all paths for speculative vectorization for the benchmarks studied in our experimental evaluation.

3. *is_path_vectorizable*: In step 3, we determine the safety of vectorizing each path selected by step 2 using existing data flow and dependence analysis techniques [36, 38, 23, 55, 40], to categorize the vectorizability of variables and statements along the path. Any path that cannot be vectorized is removed from the existing collection of paths to be considered for speculation before the final result is returned.

In step 3, *is_path_vectorizable* classifies all variables inside a path into the following categories:

- **Invariant:** Variables that are used inside the speculated path but never modified within the path. During vectorization, these scalar invariants can simply be replicated inside vectors so that the same value is used across all vectorized iterations of the selected path.
- **Local or private:** Variables that are re-initialized at each iteration of the selected path before being used along the path. During vectorization, a vector needs to be allocated for each private variable to hold the value of the variable for each vectorized iteration.
- **Recurrent:** Variables that are modified along the path after being used in the current or previous iterations of the path. Special forms of recurrent variables, e.g., loop induction and reduction variables, can be vectorized in spite of their loop-carried cross iteration dependences. However, the existence of other more general forms of recurrent variables along the speculated path would prevent the path from being vectorized.

The above categorization is made using a data flow analysis approach similar to that taken in [40], except that only the speculated path is analyzed. For instance, a variable that is modified or recurrent within a loop can be invariant or private along a speculated path, thus allowing the path to be vectorized provided that proper recovery mechanisms are in place when the speculation fails. If a path contains a recurrence that is not induced by a loop induction or reduction variable, the path is deemed unsafe to vectorize and removed from the set of paths to be considered for speculation.

To ensure each speculated path can be correctly vectorized by a later transformation step, our analysis additionally identifies all variables that belong to the following groups:

- **Live-out:** Variables that have been modified inside the vectorized path and are expected to be used after the loop terminates. For these variables, their values at the last

iteration of the loop need to be copied or reduced back to scalar variables to ensure correct references to their values after the loop is complete.

- **Live-In:** Variables that are used along the vectorizable path before they are modified within the loop. These are live-in at the entry of the loop path. For these variables, their vector representations need to be initialized with correct values before entering the vectorized path. If the variable is a reduction variable, the first element of its vector representation is initialized with its scalar value before entering the vectorized path, and the rest of the elements are initialized with zero or one (e.g., 0 is used if the reduction operation is addition, and 1 is used for multiplication reductions). Otherwise, since the safety analysis considers this path to be vectorizable, the variable is not recurrent, and all entries of its vector can simply be initialized with the scalar value before entering the vectorized path.

Figure 5.5(a) shows an implementation of the *SSQ* kernel from the BLAS library in ATLAS[69]. The if-else conditional inside this loop has generated two alternative paths to consider, as shown in Figure 5.5(b). Finally, Figure 5.5(c) illustrates our variable classification analysis for both paths: Path-1, which includes the if-branch, modifies only one non-local variable, *ssq*, through reduction, and as a result is vectorizable. On the other hand, Path-2 modifies both *ssq* and *scal* in a complex recurrent fashion and thus cannot be vectorized. By selecting Path-1 to vectorize, our speculative algorithm can partially vectorize the given input loop even though Path-2 is not at all vectorizable. To the best of our knowledge, the loop in Figure 5.5(a) is currently classified as *not vectorizable* by existing SIMD vectorization techniques.

5.1.2 Structure of Generated Code

Figure 5.6 shows the typical structure of the code generated by our vectorization transformation. Here all the statements in the original code are rearranged along two paths: the path taken when the speculative vectorization succeeds (i.e., the vectorized path), and the

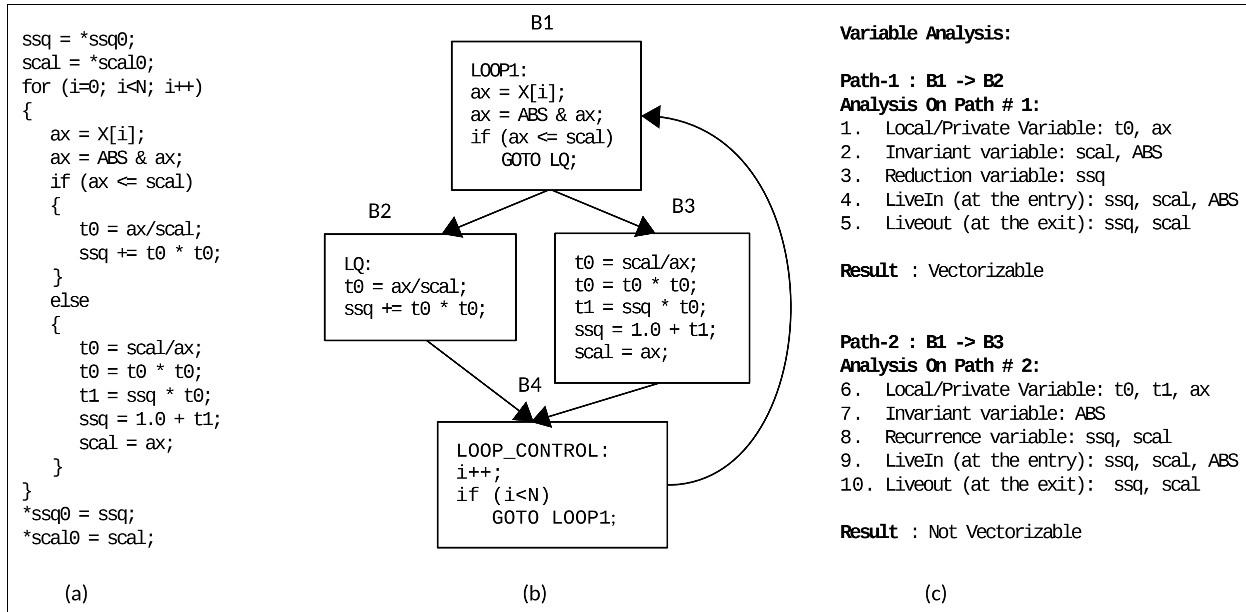


Figure 5.5: Example: analyzing the *Sum of Squares (SSQ)* kernel: (a) Scalar code (b) CFG of loop (c) Analysis

path(s) taken when the speculation fails (i.e., the scalar restart code). The vectorized path contains:

- *Vector prologue*, which contains instructions to ensure proper initialization of the vector variables for live-in and loop invariant variables;
- *Vector backup*, which contains instructions that save the values of variables that will be modified along the vectorized path so that their values can be restored in case the speculation fails;
- *Vector loop body*, which contains SIMD instructions generated from vectorizing statements and conditional jumps along the selected path;
- *Vector loop update*, which checks termination status of the vectorized loop and jumps back to the start of the vector backup section when needed;
- *Vector epilogue*, which contains instructions to properly terminate the vectorized loop by transferring the values of live-out variables back to scalar variables.

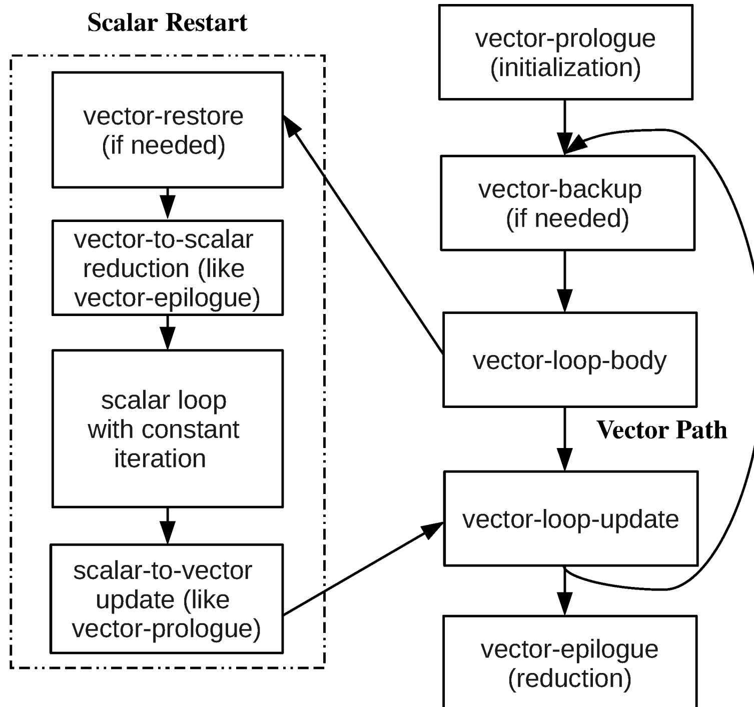


Figure 5.6: Structure of vectorized code

Although the *vector loop body* section may include many conditional exits from the speculated path, we generate only a single *scalar restart code*, which is entered whenever a speculation check fails. The restart code contains:

- *Vector restore*, which restores all variables that may have been mistakenly modified long the vectorized path, using backup variables saved during the vector-backup stage;
- *Vector-to-scalar reduction*, which copies the most current values from the *correctly modified* vector variables;
- *Constant-iteration scalar loop*, which re-evaluates the mistakenly speculated iteration of the vector loop body using scalar instructions;
- *Scalar-to-vector update*, which contains instructions that transfer the results of the scalar restart back to their respective vector variables so that future iterations can be speculatively vectorized.

```
1   VECTOR_PROLOGUE:
2   VABS = [ABS, ABS, ABS, ABS];
3   Vscal = [scal, scal, scal, scal]
4   Vssq = [ssq, 0.0, 0.0, 0.0]
5
6   VECTOR_LOOP:
7   for (i=0; i < N4; i+=4)
8   {
9   // Backup
10  // Vector Body
11      Vax = X[i:i+3];
12      Vax = VABS & Vax;
13      if (VEC_ANY(Vax > Vscal)
14          GOTO SCALAR_RESTART;
15      Vt0 = Vax/Vscal;
16      Vssq += Vt0 * Vt0;
17      continue;
18
19  /* Scalar Restart Code */
20
21      SCALAR_RESTART:
22  // Restore
23  // Vector to Scalar
24      ssq = sum(Vssq[0:3]);
25
26  // Scalar Loop
27      for (j=0; j < 4; j++)
28      {
29          ax = X[i+j];
30          ax = ABS & ax;
31          if (ax > scal)
32          {
33              t0 = scal/ax;
34              t0 = t0 * t0;
35              t1 = ssq * t0;
36              ssq = 1.0 + t1;
37              scal = ax;
38          }
39          else
40          {
41              t0 = ax/scal;
42              ssq += t0 * t0;
43          }
44      }
45
46      //Scalar to Vector Update
47      Vssq = [ssq, 0.0, 0.0, 0.0];
48      Vscal =[scal, scal, scal, scal];
49  }
50
51  VECTOR_EPILOGUE:
52  ssq = sum(Vssq[0:3]);
53  scal = Vscal[0];
```

Figure 5.7: Pseudo-code for Speculatively Vectorized SSQ Loop (our compiler generates assembly)

5.1.3 Applying the Transformation

Figure 5.7 shows an example of the speculatively vectorized loop from the original code in Figure 5.5(a). The analysis of the two paths through this loop are shown in Figure 5.5(c), where Path-1 has been selected for speculative vectorization. Figure 5.8(a) shows the initial control-flow graph for this loop, and (b)-(d) illustrate the intermediate results of our vectorization transformation. In our implementation, the speculative vectorization transformation is applied through the following five steps:

1. Speculated path formation: This step modifies the control flow of the loop body so that each conditional branch inside the speculated path (*spath*) is a potential exit from the *spath* to the unvectorized code, and all blocks that are not in the chosen path are relocated to a separate region (which will be converted to scalar restart code in step 3). This code reorganization leaves the chosen *spath* contiguous in instruction memory with the loop, increasing its spatial locality and decreasing the probability of branch mispredicts within the path. In order to make the *spath* instructions contiguous, it is necessary to reverse the branch conditionals[2] whose fall-through and goto targets are swapped by this transformation, resulting in the modified CFG shown in Figure 5.8(b). Note that blocks B2 and B3 have changed position from Figure 5.8(a).

2. Vectorization alignment and cleanup: In this step, we perform possible loop peeling in order to align vector memory access [36, 32], as well as creating a cleanup loop to handle loop iterations that are not a multiple of the vector length [4, 7, 62]. This step is not particular to speculative vectorization, and for simplicity this cleanup/alignment code is generally omitted from our figures.

3. Scalar Restart Generation: This step uses the current scalar loop to generate the scalar restart code. As shown in Figure 5.6, the scalar restart restores any possibly modified

[2]Reversing conditionals can complicate NaN handling; in our framework, like many compilers, this transformation is allowed.

recurrent variables, reduces the vector values to scalar values, and then recomputes all speculated iterations using a scalar loop, before doing scalar-to-vector initialization, and then branching back to the loop update block. At this point, the scalar restart code is complete, but the *spath* does not yet have the branch target information to reach it, which is handled in the next step, and the *spath* is not yet vectorized, which is done as the final step. In Figure 5.7, the scalar restart code is shown at lines 19-48; Here a single reduction variable, *ssq*, needs to have its scalar value restored from vectorized evaluations (line 24). Its scalar evaluation result is then later transferred back to its vector variable at line 47. A variable *scal* is modified along the scalar path at line 37 and used in the speculatively vectorized path at line 15. Therefore, its value is transferred to a vector variable at line 48 before executing the vector loop update.

4. Branch target repair and non-*spath* block removal: This step updates all conditional branch targets out of the *spath* with the label of the scalar restart code generated in the previous step. Since they are now handled by our scalar restart code, the original non-speculated path(s) from the loop are no longer referenced anywhere in the code and are therefore removed. In our example, this results in the deletion of block B3, giving rise to the CFG shown in Figure 5.8(c). At this point the control flow of the transformed code is correct, but the instructions along *spath* have not yet been vectorized, which is done by the final step.

5. *spath* Vectorization: Finally, this last step vectorizes all statements along the selected *spath* and then adds the necessary vector-prologue, vector-backup, and vector-epilogue, as outlined in Figure 5.6. In particular, all recurrent variables that may be modified before the last scalar restart exit[3] are backed up before any vectorized evaluation. In order for our speculation to be true, each conditional branch along *spath* must take the fall-through direction for all speculated iterations. Therefore, we replace each original branch comparison

[3]Speculation is proven correct after the last conditional exit from the *spath*.

with a vector comparison/test that exits to the scalar restart code if any component of the comparison failed to match our speculated result. The final CFG, including the loop cleanup, is shown in Figure 5.8(d). Figure 5.7 shows a simplified pseudo-code for the vectorized loop of our SSQ example (excludes loop peeling and loop cleanup).

5.1.4 Correctness and Generality

The main novelty of our speculative vectorization algorithm lies in the insight that when branches within a loop are strongly directional, that is, when consecutive iterations of the loop are expected to take a speculated control-flow path most of the time, SIMD vectorization can be applied to aggressively parallelize the path, with the other paths given lower priority. A similar path-based formulation has been used in trace scheduling [14], the de facto instruction scheduling algorithm widely adopted by modern compilers. However, such formulation has yet to be extended to other backend compiler optimizations beyond instruction scheduling. As far as we know, our work is the first that formulated SIMD vectorization using path-based optimization strategies.

Since our work essentially extends existing SIMD vectorization algorithms [36, 38, 23] to support control-flow path speculation and recovery, the algorithm is correct as long as the control-flow transformations are correctly performed, all the variables mistakenly modified by the speculated path can be correctly recovered, and the *spath* code branches correctly to the scalar restart code when misspeculation is detected. Our current implementation supports only speculative vectorization of a single path within a given loop, and the vectorization is disabled when the path contains memory references or variables that cannot be precisely modeled.

5.2 Integration Within iFKO

We have implemented our speculative vectorization optimization, together with several other transformations to help evaluate its effectiveness, within iFKO [62], an iterative backend compiler with an emphasis on optimizing the performance of floating-point intensive compu-

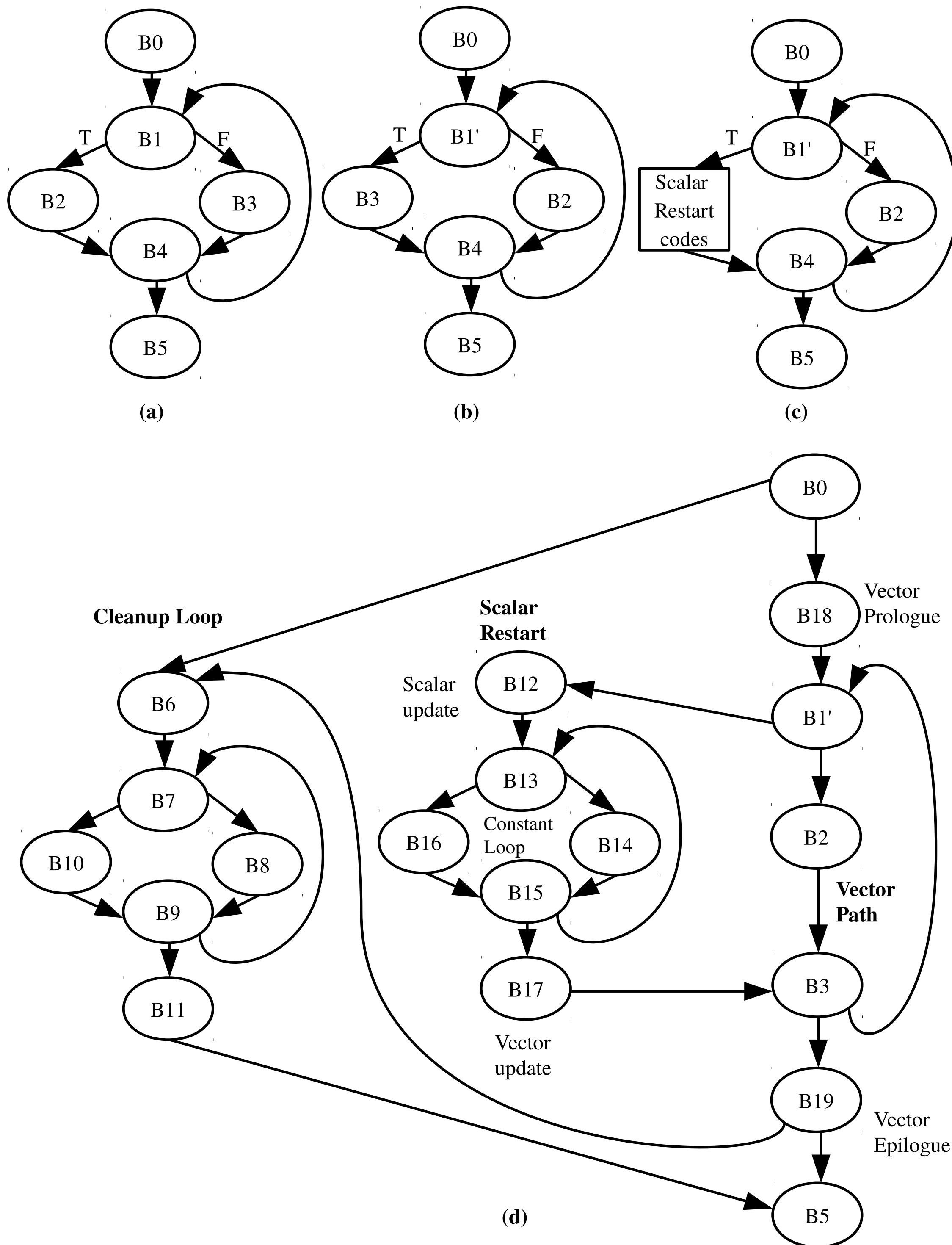


Figure 5.8: Transformation steps using Control Flow Graph: (a) Original CFG (b) CFG after step1 (c) CFG after step3 (d) CFG of speculative vectorized SSQ

tational kernels. Section 5.2.1 and 5.2.2 provide an overview of the iFKO tuning framework and the new capabilities that we have added. Section 5.2.3 discusses empirical tuning strategies we have adopted within iFKO to automatically find the fastest available vectorization method for each input kernel.

5.2.1 Overview of iFKO

iFKO [70, 62] (iterative Floating Point Kernel Optimizer) compilation framework is composed of two components, a set of search drivers that search the optimization space, and a specialized compiler called FKO that performs analysis (to determine legality of transforms as usual, but in an iterative compiler, also to bound the search space), and makes all required transformations (as discussed in the Chapter 2).

In iFKO, optimizations are split into two classes. *Fundamental transformations* are optimizations that are empirically tuned during the timing process, while *repeatable transformations* are optimizations that are repeatedly applied in series to a scope of code while they are successfully improving the code. Fundamental transforms usually have a parameter that is searched during the tuning phase. In the simplest case, the search is whether or not to apply an optimization, since it only sometimes leads to faster code. But often an optimization itself is parameterized, as in loop unrolling, where the search will find the best-performing unrolling factor in a large range. Examples of parameterized fundamental transformations include loop unrolling, prefetch distance, and accumulator expansion (see [70] for the original list of 7 fundamental transforms). Most of the repeatable transformations in iFKO are centered around optimizing register usage, see [62] for full details.

5.2.2 Extending iFKO Fundamental Transformations

In the original iFKO, SIMD vectorization was a fundamental operation with only a yes/no parameterization, as vectorization can produce a slowdown for some operations and machines. The compiler supported simple loop-based vectorization, which is enabled when the dependence distance (control & data) is greater than the vector length of the underlying

architecture. We will refer to this original vectorization method as **NHV**, for No Hazard Vectorization.

The inability of the original iFKO to apply NHV in the face of control hazards prevented it from vectorizing all of the Level 1 BLAS [70]. For this work, we added five new fundamental transformations to support vectorization past branches (some of these new optimizations help even in scalar code, as described below). These transformations are all searched by iFKO, so the best performing optimizations will be automatically selected for the user. In order to compare them in this paper, we have overridden the search using flags to require certain transformations be applied instead of searched.

We have added two new fundamental transformations that do not themselves perform vectorization, but rather transform the scalar code so that control hazards are removed, with the result that the loop can then be vectorized by NHV:

1. **MMR** (*Max/Min Reduction*): Automatically detects simple if-conditionals that serve only to compute a max or min over a sequence of values. Once found, it replaces the entire branch with the assembly MAX/MIN instruction. When MMR alone is sufficient to allow vectorization using NHV, we refer to this series of transformations leading to vectorization as **VMMR**.

2. **RC** (*Redundant Computation*): Seeks to eliminate conditional branches by replicating computations along different branches and then selecting the proper values in a fashion similar to [57]. When RC alone is sufficient to allow vectorization using NHV, we refer to this series of transformations leading to vectorization as **VRC**.

Note that in this paper we never need to apply both MMR *and* RC in order to vectorize, so this case is not discussed.

Our speculative vectorization implementation is supported by the following additional fundamental transformations:

3. **FPC** (*Frequent Path Coalescing*): Rearranges the control flow within a loop so a given path becomes a straight-line sequence of code intermixed with conditional exit jumps out of the path.

4. **SV** (*Speculative Vectorization*): If the loop targeted for vectorization has non-loop branches, examine all possible paths through the loop, and discover which are vectorizable. Our present algorithm will vectorize only one path through the loop (this simplifies our analysis & scalar restart code, but it should be possible to vectorize all legal paths with improved compilation phases). Use FPC to make the target path fall-through, and then vectorize it. All other paths are handled by scalar code.

iFKO already has a fundamental optimization called UR, which does straightforward loop unrolling. In this type of unrolling, the loop body is simply replicated as many times as requested, while avoiding moving pointers and changing loop control between unrolled iterations. We have implemented a second version of unrolling that can be used in conjunction with the existing one, so that the best performing unrolling optimization can be selected based on timing results of the optimized code.

5. **OSUR** (Over-Speculation loop UnRolling): in this type of unrolling, we speculate the path to a non-unit multiple of the vector length and inline multiple vectors of computation. This will usually pay off only for branches with very strong directional preferences, but its advantage over normal unrolling is that the overhead of speculation checking is more completely amortized by the increased speculation length. During the search, we will time OSUR & UR alone, as well as combinations of the two whenever we are tuning an SV-vectorized loop.

5.2.3 Optimization Tuning

FKO returns to the search driver a list of all possible paths through the loop. This information is then used in the following process to find the best optimized code:

- If the number of paths is one, and it is vectorizable, time both scalar code and code vectorized by NHV, and choose the best.

- If there are multiple paths through the code, choose the best performing code among:

  - Scalar code

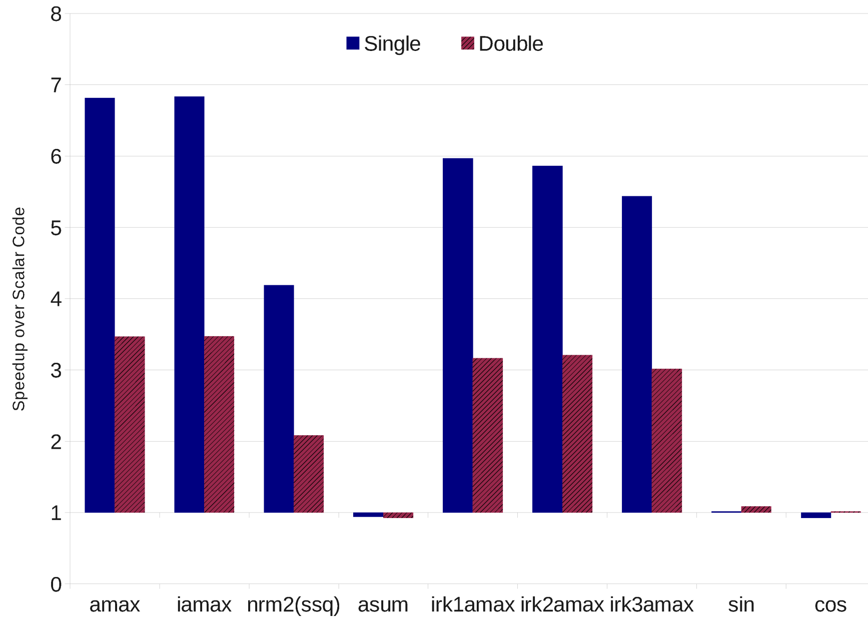
  - VMMR (if applicable)

  - VRC (if applicable)

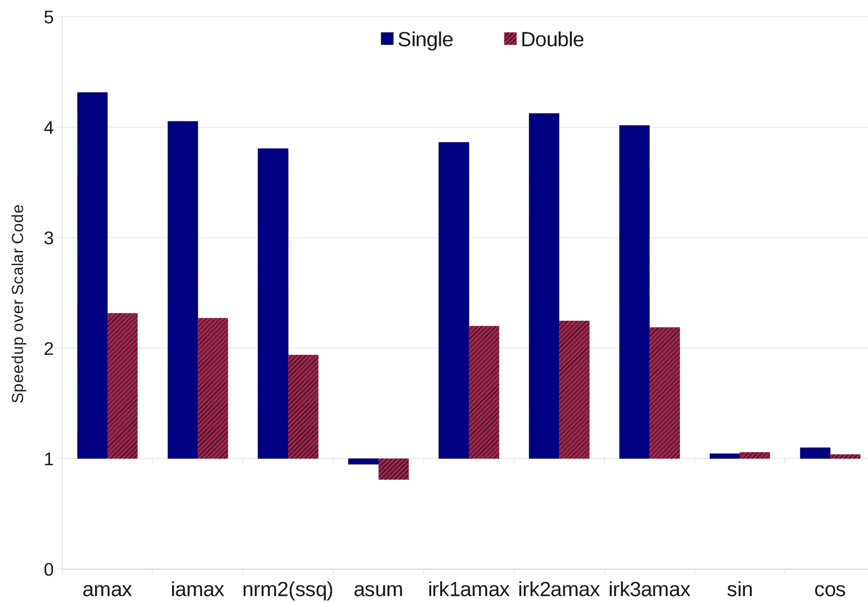
  - SV: For each path that is vectorizable, apply SV and time it, and return the best-performing code. Note that the user's timer (and its associated training data) can have a profound impact, as highlighted in Section 5.3.2.

5.3 Experiments

To validate the effectiveness of our speculative vectorization technique and the performance benefit of integrating it within an iterative optimizing compiler, we have applied the techniques to optimize 9 benchmarks, summarized in Table 5.1, with both single precision and double precision versions for each benchmark, on two machines using Intel and AMD processors respectively. The specification of the machines are listed in table 5.2. All timings utilize data chosen to fit the operands in the L2-cache, while overflowing the L1-cache; sin and all irkamax kernels utilize an 8,000 vector length input; all other kernels use a 16,000-element input to satisfy the same cache constraints. For all kernels except sin and cos, the input values use random numbers in the range [-0.5, 0.5]. For sin & cos, however, this would give our technique a strong advantage and is probably not realistic (see **S** 5.3.2 for further details). For these two kernels, we instead generate the input by passing the random values between [0, 2π] *to the wrapper functions from glib that call these kernels*; This essentially guarantees that all paths in the kernels are executed, and thus represents the worst case for our technique.



(a)



(b)

Figure 5.9: Speedup of tuned Speculative Vectorization over tuned unvectorized code for single precision (solid blue) and double precision (hatched red) : (a) Intel Corei2 (b) AMD Dozer

5.3.1 Effectiveness of Speculative Vectorization

Figure 5.9 shows the speedup speculative vectorization achieves for each benchmark over the scalar (non-vectorized) code on the Intel and the AMD machine. Both machines are using AVX, with a vector length of 8 (4) in single (double) precision. Note that both the scalar and vector versions have been empirically tuned by iFKO, so our scalar code represents the best possible case without vectorization (i.e., it is not a naive unoptimized baseline).

The first point to notice from the results is that the performance benefit of applying our vectorization technique on the Intel machine is almost twice of that on the AMD; getting peak AVX performance from the AMD Dozer is complicated by the fact that on the backend the

Table 5.1: Benchmarks used for experiments

Benchmark	Description and Library	Input Data and Size
AMAX	Absolute max value search	rand[-0.5,0.5], in-L2
IAMAX	index of absolute max, blas	rand[-0.5,0.5], in-L2
SSQ	ssq for nrm2, blas	rand[-0.5,0.5], in-L2
ASUM	Absolute sum, blas	rand[-0.5,0.5], in-L2
IRK1AMAX	Panel factorization of LU, ATLAS	rand[-0.5,0.5], in-L2
IRK2AMAX	Panel factorization of LU, ATLAS	rand[-0.5,0.5], in-L2
IRK3AMAX	Panel factorization of LU, ATLAS	rand[-0.5,0.5], in-L2
KERNEL_SIN	Kernel for sine of glibc (version:2.4,2.15)	rand[0, 2π] x on sin(); use realistic input of kernel_sin using __kernel_rem_pio2(), in-L2
KERNEL_COS	kernel for cosine of glibc (version:2.4,2.15)	rand[0, 2π] x on cos() and use realistic input of kernel_cos using __kernel_rem_pio2(), in-L2

Table 5.2: Machine specification

	Intel Corei2	AMD Dozer
Processor	Intel Xeon CPU E5-2620	AMD FX(tm)-8120
Core Speed	2.00 GHz	3.2 GHz
Total Cache Size	15360 KB	2048 KB
FPU Support	AVX(without FMA)	AVX (With FMA4)

256-bit AVX operations are split into two separate 128-bit operations, unlike on the Intel which has true 256-bits FPUs. AMD's more complex AVX handling tends to complicate scheduling on a machine that is already weak in that area, and it is also sometimes required to mix SSE and AVX instructions to maximize performance.

We expect good performance from SV only when the vectorized path is preferred. This is certainly the case for benchmarks based on max or min, which tend to change less and less frequently as the iteration count increases. These benchmarks include *amax*, *iamax*, *nrm2*, *irk1amax*, *irk2amax*, and *ir3amax*; all benefited significantly from speculative vectorization. For *asum*, SV actually causes a slowdown. Remember that for our speculation to be correct, we must correctly predict the direction of *veclen* branches, or 8 (4) branches for single (double) precision. Since the branch is on sign, and our input sign is randomly distributed, the chance of our speculation being correct is roughly $(0.5)^{veclen}$. Our speculation is almost always incorrect, and thus we continuously execute the scalar restart code. The few times our speculation is correct cannot overcome the cost of branching to the cleanup code, and we get a slowdown. Note that since our compiler can automatically select the best optimized code, SV would not be selected to optimize asum by our compiler.

Similar observations can be made for *cos* and *sin*, where multiple paths are selected based on the input data range. Single precision *cos* experiences a slight slowdown on the Intel machine, and other *cos* and *sin* results show very modest speedup. Since our speculation is almost always wrong on these kernels, the fact that we achieve any speedup at all is a measure of how low the overhead of our scalar restart code is. Of course, when iFKO is allowed to fully auto-tune codes such as this, the tuning framework will choose an alternative vectorization strategy (eg., redundant computation) or not vectorize the code at all.

5.3.2 Comparing with other Vectorization Techniques

The main strength of speculative vectorization is that it can be used in cases where the known techniques cannot be applied. In particular, if there are multiple paths through the

loop, only some of which can be successfully vectorized, SV is the only technique capable of realizing vector speeds. The NRM2 performance shown in Figure 5.9 is an example where SV allowed us to get impressive speedups when no other vectorization can be applied.

However, many kernels can be vectorized in different ways, and a compiler can always select the most promising approach based on characteristics of the input application. A reasonable heuristic can be constructed using the following line of reasoning: (1) If branches are used only for max or min, then replacing them with machine native MAX/MIN instructions (VMMR). (2) If all paths are vectorizable, and the cost of computing all sides of the branches is low, then replicate all branches to enable vectorization (VRC). (3) If a vectorizable path is strongly directional then consider speculative vectorization (SV).

Figure 5.10 shows the performance of all three vectorization methods on *amax* for the Intel Corei2. This computation is inexpensive and strongly directional, therefore a good case for both SV and VRC. We see they are both fairly competitive with VMMR, with SV performing slightly better than VRC on this machine (this essentially means that our scalar restart overhead is lower than the overhead of doing the vector compare and select). In general, we would expect that VMMR should win whenever it can be used, while the VRC and SV performance ratio will vary depending on how predictable the path is, and how much work must be performed redundantly.

Figure 5.11 compares the different vectorization methods using a *sin*, and shows how path selection can have large effects on speculative vectorization. For this benchmark in Figure 5.9 we specifically chose data in the range of $[0, 2\pi]$ which exercises all the paths in the *sin* kernel; this prevents SV from producing much speedup on the AMD system. Here we instead tune and time the code using our usual range of random inputs between [-.5, .5]. VRC is unaffected, since it is always executing code from all paths. However, this has a profound affect on SV, since it results in a particular path dominating the kernel calls made by the full sin function. As a result it goes from showing almost no speedup, to greater

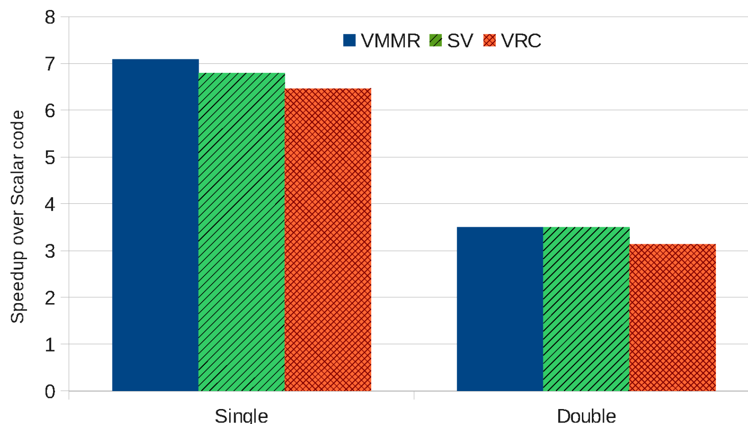


Figure 5.10: Comparison of speedups for absolute value maximum using vectorized Max/Min Reduction (VMMR, solid blue), Speculative Vectorization (SV, diagonal hashed green) and Vectorized Redundant Computation (VRC, square-hashed orange) on Intel Corei2 for both single and double precision

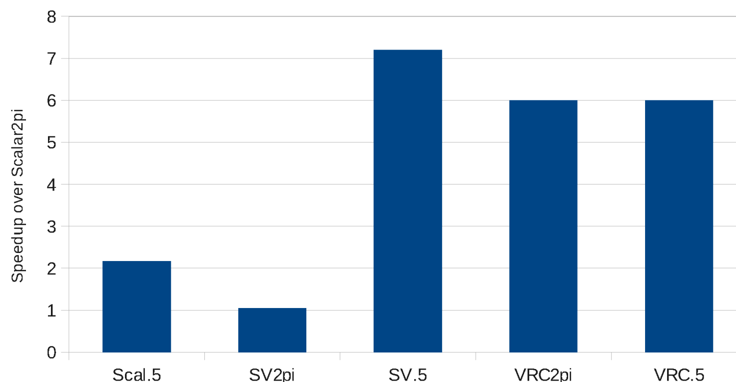


Figure 5.11: Comparison of single precision speedups on AMD Dozer for *sin* (speedups over scalar code tuned and timed for data in range [0, 2π]) using scalar code tuned and timed for data in range [-0.5, 0.5] (scal.5), Speculative Vectorization tuned and timed in range [0, 2π] (SV2pi), and range [-0.5, 0.5] (SV.5), and Vectorized Redundant Computation timed and tuned in range [0, 2π] (VRC2pi) and range [-0.5, 0.5] (VRC.5)

speedup than any other method, as SV does not perform any redundant computation and only occasionally needs to do scalar restart. Note that these speedups are inflated because we are using the speed achieved in the range of [0, 2π] as our denominator. Frequent path coalescing and related optimizations improve even the scalar code by almost a factor of 2 when we specifically tune for the [-0.5, 0.5] data range. This input sensitivity of SV is both a hazard and a meaningful opportunity for application-specific tuning for applications with known typical ranges on their data.

5.4 Related Work

The ubiquitous support of short vector operations in modern architectures has made SIMD vectorization one of the most important backend optimizations in modern compilers [34, 60, 7, 31, 57, 21, 37]. Bik et al. used *bit masking* to combine different values generated from different branches of if-else branches [7]. Shin, Hall, and Chame [57] managed dynamic control flow inside vectorized code through predicated execution of vectorized instructions and have implemented their schemes using mask and select vector operations. The technique was later improved to bypass some of the redundant vector computations for complex nested control flows [56]. Karrenberg et al. [30] presented a similar approach but introduced the mask and select operations in the SSA form to handle arbitrary control flow graphs. Our work also aims to enhance the effectiveness of automatic vectorization in the presence of complex control flow. Our techniques, however, focus on speculatively vectorizing strongly biased control-flow paths that are expected to be taken frequently at runtime. Our vectorization algorithm is based on existing loop-based vectorization techniques [7, 60, 21], but the path speculation strategy can be used to enhance superword-level vectorization frameworks [**?**] in a similar fashion.

Speculation is an approach commonly used in compilers when facing unknown control or data flow that prevent effective optimization [20, 35], e.g., instruction scheduling [20, 22] and thread-level parallelization [48, 15, 19]. Pajuelo et al. [41] proposed micro architecture

extension to apply vectorization speculatively. To the best of our knowledge, our work is the first that uses path-based speculation to enhance the effectiveness of SIMD vectorization within compilers.

5.5 Conclusions and Future Work

This chapter presents a new technique, speculative vectorization, which extends existing SIMD vectorization techniques to aggressively parallelize statements embedded inside complex control flow by speculating past dependent branches and selectively vectorizing paths that are expected to be taken frequently at runtime. We have implemented our technique inside the iterative backend optimizing compiler, iFKO, and have applied the path-based speculative vectorization approach to optimize 9 floating point kernel benchmarks. Our results show that up to 6.8X speedup for single precision and up to 3.4X speedup for double precision can be attained for these benchmarks in AVX through our speculative vectorization optimization. Our formulation allows partial vectorization of computations in the presence of complex control flow beyond what has been supported by existing known SIMD vectorization techniques.

Our speculation approach is complimentary and can be applied to enhance the effectiveness of most existing SIMD vectorization techniques. In future work, we will investigate applying path speculation in conjunction with known techniques. For instance, in kernels with multiple branches inside the loop, it may make sense to eliminate some branches with redundant computation, while speculating past others, and this may lead to much greater speedups than either technique can achieve when applied in isolation. A related idea is to speculate more than one path for kernels possessing more than one vectorizable path.

As vector lengths continue to grow, it may become increasingly unlikely that a branch will go in the same direction for the entire vector length for many kernels (branches such as underflow/overflow guards should be unaffected by increasing length). For kernels where increasing vector lengths are problematic, we will need to investigate *underspeculation*, where

we speculate to only some fraction of the vector length. This is a classic trade off where increased speculation accuracy reduces peak SIMD performance; by using empirical tuning we can find the most effective trade off, whether that is full, under-, or over-speculation.

Another technique that should be complementary with speculative vectorization is an adaptation of loop specialization, where we maintain the original scalar loop in the code along with the speculatively vectorized loop, and, if at runtime we detect too many jumps to the scalar cleanup code, we switch to the unvectorized code for the rest of the computation. The only thing that we would need to add to our framework to support this is scalar restart counting and some generalization of our loop specialization code, which should be straightforward.

CHAPTER 6
SLP VECTORIZATION IN FKO

ATLAS does not directly tune the full matrix multiply BLAS API, GEMM (GEneral rectangular Matrix Multiply), instead it tunes simpler microkernel (*gemmμ*) that operates on matrices that have been copied to a format optimized for high performance access. The framework is capable of auto-tuning a suite of *gemmμ* with individual kernels optimized for particular problem dimensions. To vectorize the kernel suite, ATLAS's generator currently uses the SIMD vector intrinsics supported by various compilers. Given this, why are we interested in autovectorize scalar code in addition to use intrinsics? The reasons are as follows:

- Even though we currently target sophisticated users, we eventually want to evolve FKO until it can deliver excellent performance for non-computational experts. We therefore want to use the ATLAS kernel set as a starting point for this evolution. FKO can already auto-vectorize simple loop-carried vectorization using no-hazard loop-vectorization as implemented in [62]. We have since extended auto-vectorization in FKO so that it can optimize in face of branches using speculative vectorization (SV) [61] along with no-hazard after applying if conversion with redundant computation. Now we need some way to find vectorization for computations that cannot even be expressed as rolled loops, as in ATLAS's access-major *gemmμ* kernels.

- As an extensible method to find arbitrary SIMD parallelism, the best state of the art method we found was the superword level parallelization (SLP) [31]. SLP is done at block level for generality, but a state of the art way of extending SLP to arbitrary loop nests has yet to emerge. Block level parallelism is insufficient when one targets hand-tuned levels of performance. For HPC usage even low order terms (outer loops) cannot be ignored. Therefore, the state of the art SLP is insufficient for our usage.

Further, existing compliers (e.g., ICC, GCC, LLVM) all failed to get good performance on our access-major formats unless we provide intrinsic code (see the Section 6.2). This motivated our SLP extension as outlined in the following sections.

- In order to achieve maximum performance, intrinsics must be tied to architecture specifics. Generalizations of intrinsics tend to lose performance on some architecture while code using outdated intrinsics may prove inflexible even if the compiler supports additional SIMD architectural features. Therefore, once the compiler knows of new architectural features, auto-vectorized code can exploit it whereas an intrinsic implementation may not be able to be improved by the compiler.

ATLAS's *gemmμ* kernels come in two variants: MVEC (vectorized along the rows of the output matrix C) and KVEC (vectorized along K, the dimension common to both input matrices A and B). Both of these variants have three levels of loop nesting and the loop order is MNK. Both are unrolled and jammed. u_m, u_n and u_k represent the unroll factor of M-, N-, and K-loops, respectively. Appendix A and B show the full listings of the *gemmμ* kernels in FKO's input language (HIL) and in C that we will later use in our results section. Figure 6.1 shows the loop-nests of both gemmμ kernel types. In Figure 6.1(a), we have loop-nests of MVEC kernel with unroll factor, $u_m = 4$, $u_n = 4$ and $u_k = 1$ and in (b), we show loop-nests of KVEC kernel with unroll factor $u_m = 4$, $u_n = 1$, $u_k = 4$.

6.1 Description of SLP in FKO

Superword Level Parallelism (SLP)[31] is the state of the art method for auto-vectorizing the straight line code in any basic block. The main idea of SLP is to exploit ILP by scheduling isomorphic statements (statements which contain the same operations in the same order) to pack them together into vector operations. SLP vectorization can also be used to vectorize the inner-most loop. Later work [39] has extended SLP to vectorize loops after unrolling. However, auto-vectorization of multiple loop-nests with SLP is harder. We have implemented a special strategy for SLP in FKO to support nested loops for the ATLAS kernels. In the

```
for (i=0; i < nmus; i++)
{
   for (j=0; j < nnus; j++)
   {
      rA0 = pA[0];
      rA1 = pA[1];
      rA2 = pA[2];
      rA3 = pA[3];
      rB0 = pB[0];
      rC00 = rA0 * rB0;
      rC10 = rA1 * rB0;
      rC20 = rA2 * rB0;
      rC30 = rA3 * rB0;
      rB1 = pB[1];
      rC01 = rA0 * rB1;
      rC11 = rA1 * rB1;
      rC21 = rA2 * rB1;
      rC31 = rA3 * rB1;
      rB2 = pB[2];
      rC02 = rA0 * rB2;
      rC12 = rA1 * rB2;
      rC22 = rA2 * rB2;
      rC32 = rA3 * rB2;
      rB3 = pB[3];
      rC03 = rA0 * rB3;
      rC13 = rA1 * rB3;
      rC23 = rA2 * rB3;
      rC33 = rA3 * rB3;
      PA += 4;    pB += 4;
      for (k=1; k < K; k++)
      {
         rA0 = pA[0];
         rA1 = pA[1];
         rA2 = pA[2];
         rA3 = pA[3];
         rB0 = pB[0];
         rB1 = pB[1];
         rB2 = pB[2];
         rB3 = pB[3];
         rC00 += rA0 * rB0;
         rC10 += rA1 * rB0;
         rC20 += rA2 * rB0;
         rC30 += rA3 * rB0;
         rC01 += rA0 * rB1;
         rC11 += rA1 * rB1;
         rC21 += rA2 * rB1;
         rC31 += rA3 * rB1;
         rC02 += rA0 * rB2;
         rC12 += rA1 * rB2;
         rC22 += rA2 * rB2;
         rC32 += rA3 * rB2;
         rC03 += rA0 * rB3;
         rC13 += rA1 * rB3;
         rC23 += rA2 * rB3;
         rC33 += rA3 * rB3;
         PA += 4;
         PB += 4;
      }
      pC[0] = rC00; pC[1] = rC10;
      pC[2] = rC20; pC[3] = rC30;
      pC[4] = rC01; pC[5] = rC11;
      pC[6] = rC21; pC[7] = rC30;
      pC[8] = rC02; pC[9] = rC12;
      pC[10] = rC22; pC[11] = rC32;
      pC[12] = rC03; pC[13] = rC13;
      pC[14] = rC23; pC[15] = rC33;
      PC += 16;
      PA = pA0;
   }
   PB = pB0;
   PA0 += incAM;
   PA = pA0;
}
```

(a)

```
for (i=0; i < nmus; i++)
{
   for (j=0; j < nnus; j++)
   {
      rC00 = 0.0;
      rC10 = 0.0;
      rC20 = 0.0;
      rC30 = 0.0;

      for (k=0; k < K; k+=4)
      {
         rB0 = pB[0];
         rA0 = pA[0];
         rC00 += rA0 * rB0;
         rA0 = pA[4];
         rC10 += rA0 * rB0;
         rA0 = pA[8];
         rC20 += rA0 * rB0;
         rA0 = pA[12];
         rC3_0 += rA0 * rB0;

         rB0 = pB[1];
         rA0 = pA[1];
         rC00 += rA0 * rB0;
         rA0 = pA[5];
         rC10 += rA0 * rB0;
         rA0 = pA[9];
         rC20 += rA0 * rB0;
         rA0 = pA[13];
         rC30 += rA0 * rB0;

         rB0 = pB[2];
         rA0 = pA[2];
         rC00 += rA0 * rB0;
         rA0 = pA[6];
         rC10 += rA0 * rB0;
         rA0 = pA[10];
         rC20 += rA0 * rB0;
         rA0 = pA[14];
         rC30 += rA0 * rB0;

         rB0 = pB[3];
         rA0 = pA[3];
         rC00 += rA0 * rB0;
         rA0 = pA[7];
         rC10 += rA0 * rB0;
         rA0 = pA[11];
         rC20 += rA0 * rB0;
         rA0 = pA[15];
         rC30 += rA0 * rB0;

         PA += 16;
         PB += 4;
      }
      pC[0] = rC00;
      pC[1] = rC10;
      pC[2] = rC20;
      pC[3] = rC30;
      PC += 4;
      PA = pA0;
   }
   PB = pB0;
   PA0 += incAM;
   PA = pA0;
}
```

(b)

Figure 6.1: Loopnests of access major martix-matrix multiplication(AMM) kernels: (a) MVEC AMM kernel with $u_m = 4$, $u_n = 4$, $u_k = 1$ (b) KVEC AMM kernel with $u_m = 4$, $u_n = 1$ and $u_k = 4$

following sections we will describe how our SLP vectorization works for the *gemmμ* loop-nests. We will first describe our SLP vectorization for a single basic block, then illustrate how we extend it to vectorize the innermost loop and eventually, vectorize whole loop-nests from this starting point. We have also implemented a hybrid SLP technique where we can use different vectorization techniques for the innermost loop and extend vectorization towards outer loops (discussed in Section 6.1.5).

6.1.1 Basic Block Vectorization

Our SLP implementation for single basic block works mostly like the original SLP [31]. Therefore our single basic block SLP performs following three steps:

1. Create initial/seed packs: In FKO, we have created initial packs of statements usually either by grouping of vector length's numbers of adjacent memory loads or stores. However, unlike the original SLP, initial packs can also be formed by the vectors created in predecessor (successor) basic blocks if they are live-in (live-out) to this block.

2. Extend packs from initial packs using def-use and use-def chain: Once the initial packs of statements have been created, FKO can extend the packs with independent isomorphic instructions by following the def-use and use-def chains [1]. The idea here is to find new candidates that can either (a) produce needed source operands in existing packs by using use-def chain or (b) use the operands defined in existing packs as the source operands by using def-use chain. The order of the packs in the initial set is important as well.

3. Schedule packs and emit vector instructions: Now that we have all the candidate packs created, we need to schedule the statements of the basic block to map the statements of candidate packs. FKO performs dependence analysis before scheduling statements to map the packs to ensure that statements in packs can be executed safely in parallel. FKO starts scheduling statements based on the order of the statements in the block. While scheduling a statement in the block, FKO tries to schedule all statements of the pack which that statement belongs as long as all the statements on which it is dependent on have been scheduled. If

the scheduling is successful for the whole block, vector statements can be emitted for each group of such statement in pack[1].

Figure 6.2 shows how FKO creates and extends packs (step 1 and step 2) for the basic block of the innermost loop (kloop) of the MVEC4x4x1 kernel shown in Figure 6.1(a). Figure 6.3(a) shows the output after scheduling (step 3) the code and Figure 6.3(b) shows the final vectorized code. In the kloop block of Figure 6.2(a), we rearranged some of the computational statements to aid in our description of SLP scheduling. As a first step, FKO creates initial/seed packs from the adjacent memory loads of pA (see Figure 6.2(b)) and that of pB (see Figure 6.2(d)). Note that we assume four as the length of SIMD vectors in this example giving us four statements per pack. Let us first assume that the seed pack $P0$ in Figure 6.2(b) (based on loads of `pA`) is selected. In this case, the tuple of variables (`rA0`, `rA1`, `rA2`, `rA3`) is set/defined in this pack, and FKO finds four additional packs ($p1$ to $p4$ in Figure 6.2(c)) by exploring the def-use chain where this tuple of variables is used. But if FKO uses the seed of Figure 6.2(d) (based on loads of `pB`), it would explore the packs shown in Figure 6.2(e) where we have tuple of (`rB0`, `rB1`, `rB2`, `B3`) variables. So, based on the order of exploration of the initial packs, SLP can result different vectorized codes. In FKO, we evaluate all those different valid vector codes and estimate the best one by the effectiveness of the vectorization in terms of the entire loop-nests. The deeper the loopnest is, the higher the weight it has. Note that once the seed pack of Figure 6.2(b) is selected and used to explore the packs, FKO cannot use the pack of Figure 6.2(d) since the sequence of tuple (`rB0`,..,`rB3`) violates the required sequence of tuple (e.g., (`rB0`, `rB0`, `rB0`, `rB0`)) in selected $P1$ to $P4$ packs (see Figure 6.2(c)). FKO then schedules the statements of the basic block of the kloop (shown in Figure 6.3(a)) considering the packs in Figure 6.2(b) and 6.2(c). Note that the schedule of statements that the packs dictate may not be legal due to the dependence

[1]In FKO, scheduling and vectorization are done simultaneously but on copied basic blocks. If scheduling is not successful, all those copies are deleted without changing the original code.

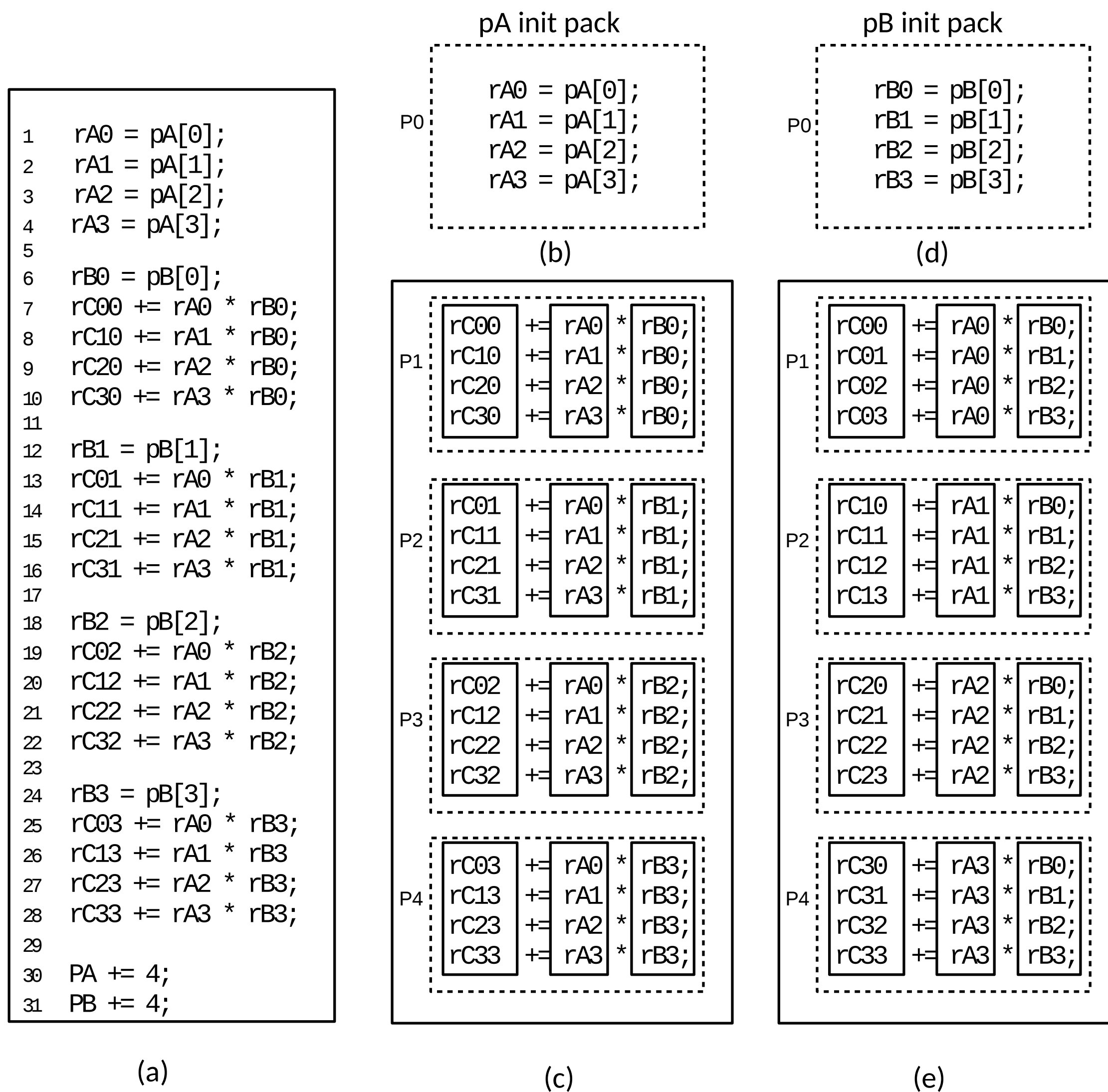


Figure 6.2: Pack creation in SLP: (a) possible innermost loop (kloop) of MVEC4x4x1 kernel, (b) initial packs based on loads of pA (c) pack extension based on pA init pack (d) initial pack based on loads of pB, (e) pack extension based on pB init pack

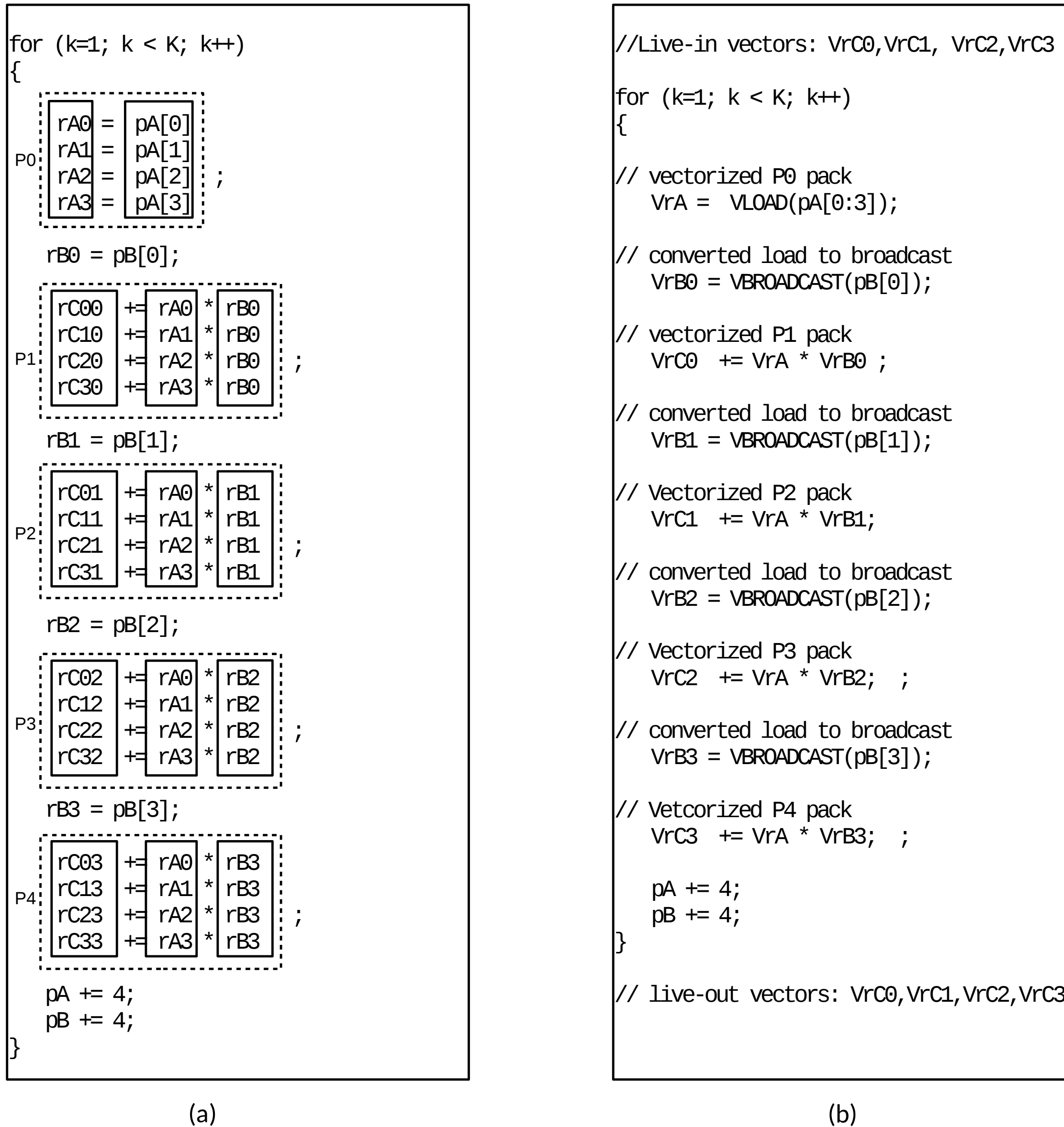


Figure 6.3: SLP vectorization of kloop for MVEC4x4x1: (a) After scheduling the code based on packs (b) After emitting vector codes

among the statements in the block. FKO therefore performs dependence analysis at the beginning of this step. The statement in line 1 of Figure 6.2(a) is part of the seed pack in Figure 6.2(b). Therefore, all the four statements of that pack have been scheduled together (see Figure 6.4(a)). They are already in order and therefore do not violate the dependence. Figure 6.3(b) shows how `VrA` vector is created and the loads of pA are converted into a vector load of pA from this pack when the vector code is emitted. The memory load of pB in line 6

of Figure 6.2(a) has been initially scheduled as standalone scalar instruction since we are not using pack $P0$ of Figure 6.2(d). The statement in line 7 of Figure 6.2(a) is part of pack $P1$ in Figure 6.2(c), we therefore want to schedule all the statements in $P1$ (located from line 7 to line 10) consecutively. They are in order and do not create any dependence issue. However, to emit vector statement, we need to create vectors for all the source and destination tuples of variables guided by the pack. Note that `VrA` (`rA0`, `rA1`, `rA2`, `rA3`) is already created. `VrC0` is created with the tuple (`rC00`, `rC10`, `rC20`, `rC30`) and marked as live-in since all the scalars in the tuple are live-in at the entry of the block. When `VrB0` is created with the tuple (`rB0`, `rB0`, `rB0`, `rB0`), the load statement of `pB` in line 6 is converted into vector broadcast (see Figure 6.3(b)) when the architecture supports such instruction (otherwise, shuffle instructions are used to initialize `VrB0` from `rB0`). FKO similarly schedules and emits vector instructions for the rest of the statements in this block.

6.1.2 Innermost Loop Vectorization

SLP can be used to vectorize the innermost loop after unrolling the loop by vector length iterations, as a more general approach to unrolling for vectorization discussed in [39] as loop-aware SLP. The idea here is to unroll the loop by certain iterations and then pack the statements similar to the above process. In addition to loop unrolling, FKO applies scalar renaming and scalar expansion to resolve the dependences. Figure 6.1(b) shows a KVEC kernel with $u_m = 4$ $u_n = 1$ and $u_k = 4$. This kernel is already unrolled by four iterations in order to expose KVEC access major storage. FKO recognizes the pattern of reduction variables (`rC00`, `rC10`, `rC20` and `rC30`) of this kernel. It then applies accumulator expansion to create four accumulators for each of `rCx0`, and then it renames all the local variables. The output of such transformations of loop is shown in Figure 6.4(a). Note that due to accumulator expansion, each instance of `rCx0` is replaced by one of the four expanded variables (e.g., `rC00_0`,`rC00_1`, etc.). They are initialized before the loop and accumulated back in the original variables after the loop. After performing these optimizations, our single

basic block SLP vectorizer can efficiently vectorize the entire loop block. The vectorized pseudo-code for the innermost loop of KVEC4x1x4 kernel shown in Figure 6.1(b) is given in Figure. 6.4(b). Note that we would need vector-initialization at the end of the preheader of the innermost loop to create vectors from scalar variables which are live-in at the entry of the loop and vector-reduction in the posttail to reduce the vector into a scalar for those who are live-out at the end of the loop. However, we can optimize the reduction of vectors further using a special sequence of vector codes, as discussed in the next section.

6.1.2.1 Optimizing Vector Reductions With the VVRSUM Computation

It is common to apply scalar expansion before vectorization to increase the dependence distance of reduction variables both in no-hazard and SLP loop vectorization. Therefore, we may end up with reduction codes after the loop which are used to reduce scalar expanded variables into the original reduction variables. These reduction codes are inherently non-vectorizable since they require operations within a given vector (which we call horizontal operation). When we perform horizontal operation, the parallelism of the vector computation is logarithmically reduced. However, we can maintain full parallelism throughout the operation if we always do reductions in batches of the vector length (i.e., we produce a vector of reduced accumulators by reducing *veclen* accumulators at once).

For accumulator reduction, we have a specialized series of instructions called VVRSUM to perform *veclen* reductions in a fully parallel fashion. For each architecture, the details of VVRSUM can be different depending on the supported operations. We illustrate it here for x86, where we can implement this technique using the horizontal add instruction of SSE and AVX. Figure 6.5 shows how this VVRSUM can be implemented in SSE. As the input we have four vectors whose horizontal sum we need. The output of VVRSUM is another vector which consists of the reduced-sum of each input vectors. The sequence of the elements of the output vector depends on the sequence of input vectors. The implementation of VVRSUM in SSE uses only three `HADD` instructions for four vectors with four elements. However, it is

```
// initialize scalar expanded vars
rC00_0 = rC00_1 = rC00_2 = rC00_3 = 0.0;
rC10_0 = rC10_1 = rC10_2 = rC10_3 = 0.0;
rC20_0 = rC20_1 = rC20_2 = rC20_3 = 0.0;
rC30_0 = rC30_1 = rC30_2 = rC30_3 = 0.0;

for (k=0; k < K; k+=4)
{
   rB0_0 = pB[0];
   rA0_0 = pA[0];
   rC00_0 += rA0_0 * rB0_0;
   rA0_1 = pA[4];
   rC10_0 += rA0_1 * rB0_0;
   rA0_2 = pA[8];
   rC20_0 += rA0_2 * rB0_0;
   rA0_3 = pA[12];
   rC30_0 += rA0_3 * rB0_0;

   rB0_1 = pB[1];
   rA0_4 = pA[1];
   rC00_1 += rA0_4 * rB0_1;
   rA0_5 = pA[5];
   rC10_1 += rA0_5 * rB0_1;
   rA0_6 = pA[9];
   rC20_1 += rA0_6 * rB0_1;
   rA0_7 = pA[13];
   rC30_1 += rA0_7 * rB0_1;

   rB0_2 = pB[2];
   rA0_8 = pA[2];
   rC00_2 += rA0_8 * rB0_2;
   rA0_9 = pA[6];
   rC10_2 += rA0_9 * rB0_2;
   rA0_10 = pA[10];
   rC20_2 += rA0_10 * rB0_2;
   rA0_11 = pA[14];
   rC30_2 += rA0_11 * rB0_2;

   rB0_3 = pB[3];
   rA0_12 = pA[3];
   rC00_3 += rA0_12 * rB0_3;
   rA0_13 = pA[7];
   rC10_3 += rA0_13 * rB0_3;
   rA0_14 = pA[11];
   rC20_3 += rA0_14 * rB0_3;
   rA0_15 = pA[15];
   rC30_3 += rA0_15 * rB0_3;


   pA += 16;
   pB += 4;
}

// reduce scalar expanded vars
rC00 = rC00_0 + rC00_1 + rC00_2 + rC00_3;
rC10 = rC10_0 + rC10_1 + rC10_2 + rC10_3;
rC20 = rC20_0 + rC20_1 + rC20_2 + rC20_3;
rC30 = rC30_0 + rC30_1 + rC30_2 + rC30_3;
```

(a)

```
// vector initialization
… … …

for (k=0; k < K; k+=4)
{
  rB0_0
  rB0_1
  rB0_2  = VLOAD(pB[0:3]);
  rB0_3

  rA0_0
  rA0_4
  rA0_8  = VLOAD(pA[0:3]);
  rA0_12

  rC00_0     rA0_0     rB0_0
  rC00_1     rA0_4     rB0_1
  rC00_2  += rA0_8  *  rB0_2  ;
  rC00_3     rA0_12    rB0_3

  rA0_1
  rA0_5
  rA0_9  = VLOAD(pA[4:7]);
  rA0_13

  rC10_0     rA0_1     rB0_0
  rC10_1     rA0_5     rB0_1
  rC10_2  += rA0_9  *  rB0_2  ;
  rC10_3     rA0_13    rB0_3

  rA0_2
  rA0_6
  rA0_10  = VLOAD(pA[8:11]);
  rA0_14

  rC20_0     rA0_2     rB0_0
  rC20_1     rA0_6     rB0_1
  rC20_2  += rA0_10 *  rB0_2  ;
  rC20_3     rA0_14    rB0_3

  rA0_3
  rA0_7
  rA0_11 = VLOAD(pA[12:15]);
  rA0_15

  rC30_0     rA0_3     rB0_0
  rC30_1     rA0_7     rB0_1
  rC30_2  += rA0_11 *  rB0_2  ;
  rC30_3     rA0_15    rB0_3

 pA += 16;
 pB += 4;
}
// vector reduction
… … …
```

(b)

Figure 6.4: SLP vectorization for kloop of KVEC4x1x4 : (a) kloop after renaming and accumulator expanding (c) kloop after vectorization (showing elements of vector inside box)

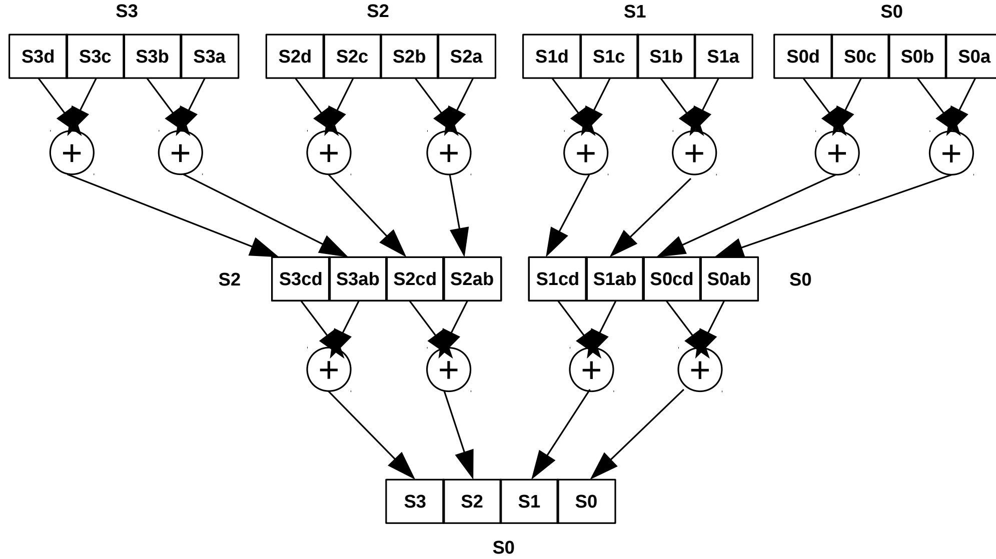


Figure 6.5: SSE implementation of VVRSUM using $veclen = 4$ and four accumulator expanded accumulators

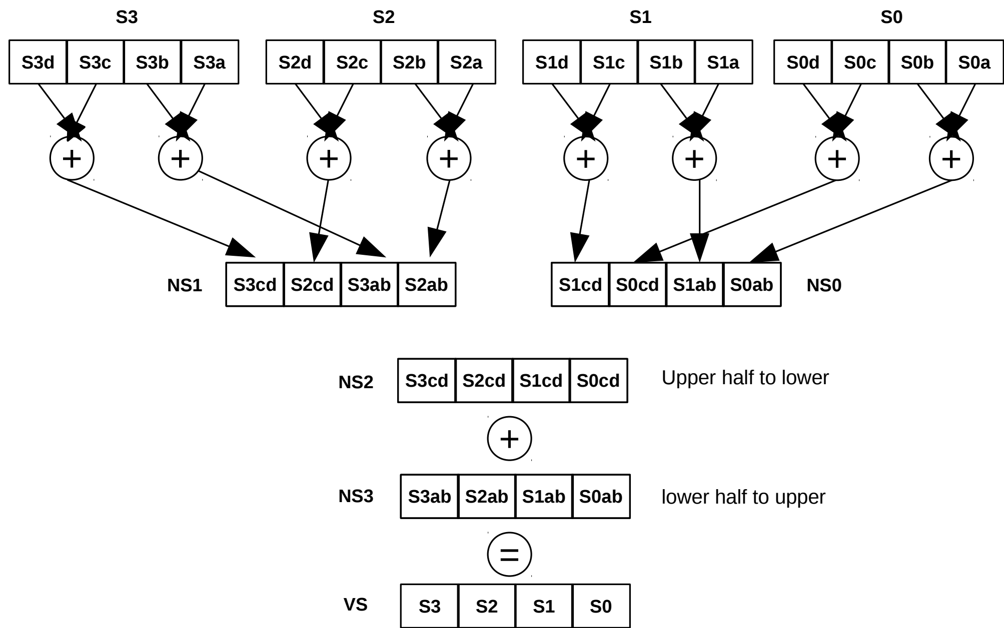
more tricky in AVX since the `HADD` operation works differently. Figure 6.6 shows how we can implement VVRSUM in AVX machine. As shown in the figure, we need extra two shuffle instructions to fix the order of the elements after performing two horizontal operations. Then we need to perform a vertical operation to sum them up.

6.1.3 Loop-nests Vectorization

Vectorizing loops is most challenging and is also an essential optimization for our studied *gemmµ* kernels. To simplify the problem with loop-nests, we define simple loop-nests body in Definition 1.

Definition 1 (Simple Loop Nests Body)**.** *LOOPi = LOOPi-1 + Preheader + Posttail; LOOP0 = Single Loop block;*

Definition 1 is a recursive definition of the body of such simple loop-nests which FKO applies SLP on. At the depth-most level, the innermost loop consists of a single basic block. The immediate upper-level loop of the innermost loop consists of the innermost loop and its



AVX Implementation (double precison)

Pseudocode:	**Example:**
NS0 = HADD(S1, S0);	haddpd xmm0, xmm1, xmm4
NS1 = HADD(S3, S2);	haddpd xmm2, xmm3, xmm5
NS2 = UPPERHALF2LOWER(NS1, NS0)	vpermute2f128 0x31, xmm5, xmm4,xmm6
NS3 = LOWERHALF2UPPER(NS1, NS0)	vpermute2f128 0x20, xmm5, xmm4,xmm7
VS = VADD(NS3, NS2);	vaddpd xmm6, xmm7, xmm8

Figure 6.6: AVX implementation of VVRSUM

preheader and posttail only. The preheader is the block which itself is outside of the loop and has only the header as successor, and so all paths to the header from outside the loop reach the header through preheader [1]. Post-tail, on the other hand, is the successor of the tail of the loop. The posttail is outside the loop and has no non-tail predecessor. In our definition of the body of simple loop-nests, we have exactly one preheader and posttail for any loop. In other word, our loop-nest does not have any branches other than the back edge of the loop. If any loop-nest has any branch depending on the condition of loop-invariant variables, we first apply loop unswitching to pull the branches out of the loop-nests. We then apply our SLP vectorization method for the loop-nests. Figure 6.7(a) shows the CFG of a loopnest matching the Definition 1. We use the SLP vectorization of single block as a building block to vectorize the whole loop nest. Unlike SLP in other compilers (e.g., SLP in LLVM), we start vectorizing our loop-nests from the innermost loop (provided that this loop has adjacent memory accesses). We don't limit our SLP to a start pack only with memory stores. We can start initializing our SLP packs from adjacent memory loads, stores or predefined vectors passed as an argument (especially important when vectorizing the preheader and posttail of a loop). In the case of innermost loop, we use memory loads and/or stores to initialize the packs similar to our previous discussion in Section 6.1.1. However, for the preheader and posttail blocks of a loop, we first try to initialize the packs based on the live-in and live-out vectors of the loop. If we don't have live-in/live-out packs or can't form packs from them, we then try to form packs from memory loads/stores as outlined in Section 6.1.1.

For example, in Figure 6.7(c) we first try to initialize pack for B1 using the live-in vectors of B0 and for B2 using the live-out vectors of B0. Eventually, for B5 block, we try to form initial packs from the live-in vectors of B3 and for B6, we use live-out vectors of B4. the rest of the SLP steps are the same as the steps of SLP for single block basic. However, in the posttail block, reduction codes often must be added due to the scalar expansion in innermost loop and they can be efficiently parallelized using VVRSUM (as discussed in

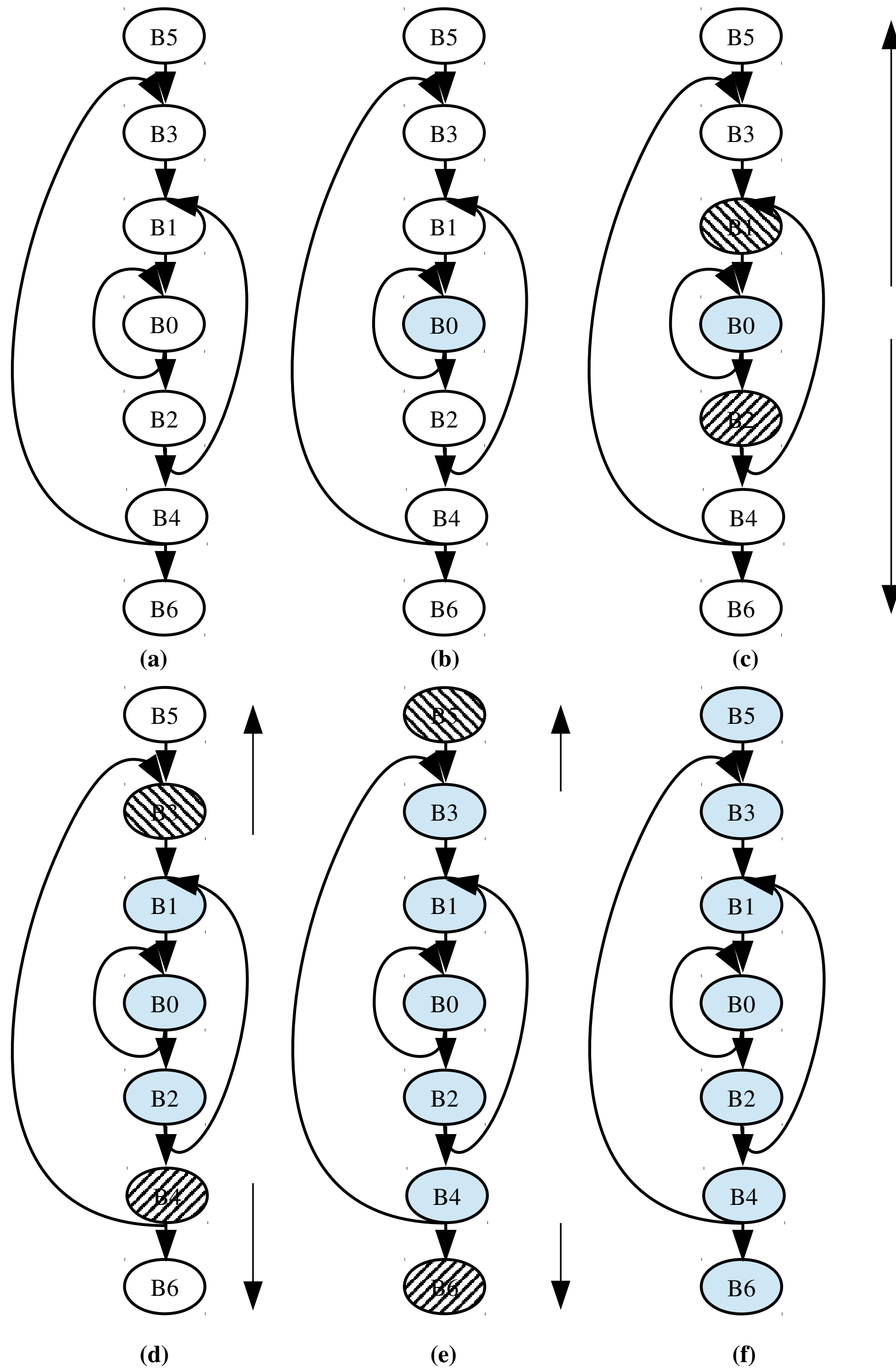


Figure 6.7: Steps in loop nests vectorization

```
//end of loop
//start of post-tail

 rC00 = rC00_0 + rC00_1 + rC00_2 + rC00_3;
 rC10 = rC10_0 + rC10_1 + rC10_2 + rC10_3;
 rC20 = rC20_0 + rC20_1 + rC20_2 + rC20_3;
 rC30 = rC30_0 + rC30_1 + rC30_2 + rC30_3;

 pC[0] = rC00;
 pC[1] = rC10;
 pC[2] = rC20;
 pC[3] = rC30;

 pC += 4;
 pA = pA0;
```

(a)

```
//end of loop
//start of post-tail
//reduction code deleted

 pC[0] = rC00;
 pC[1] = rC10;
 pC[2] = rC20;
 pC[3] = rC30;

 pC += 4;
 pA = pA0;
```

(b)

```
//end of loop
//start of post-tail
//reduction code deleted
//remaining code vectorized

                rC00
                rC10
 pC[0:3] =      rC20   ;
                rC30

 pC += 4;
 pA = pA0;
```

(c)

```
//end of loop
//start of post-tail
//VVRSUM code added

 rC00             rC00_0  rC10_0  rC20_0  rC30_0
 rC10             rC00_1  rC10_1  rC20_1  rC30_1
 rC20  = VVRSUM(  rC00_2  rC10_2  rC20_2  rC30_2  );
 rC30             rC00_3  rC10_3  rC20_3  rC30_3

                rC00
                rC10
 pC[0:3]  =     rC20   ;
                rC30

 pC += 4;
 pA = pA0;
```

(d)

Figure 6.8: SLP vectorization for posttail of kloop of KVEC4x1x4 : (a) posttail after accumulator expansion (b) posttail after deleting reduction code (c) posttail after vectorizing remaining codes (d) posttail after adding vvrsum codes at the top

Section 6.1.2.1). Hence, to vectorize the posttail of the loop, we delete the reduction codes from the block, and vectorize the rest of the statements of the block using our single block SLP vectorization. Based on the packs of SLP, we use a special sequence of live-in vectors as the input vectors to VVRSUM to generate appropriate output which matches the vectors of the posttail. Figure 6.8 shows how FKO vectorizes the posttail of the loop using VVRSUM code sequences (shown as a function). FKO deletes the reduction codes (see Figure 6.8(b)) at the beginning of the posttail block in Figure 6.8(a) to replace them with VVRSUM codes later. FKO then applies single block SLP to rest of the code (see Figure 6.8(c)). Based on the adjacent memory store pattern, it creates pack with memory store statements and vectorizes

the codes. FKO then adds VVRSUM codes with appropriate input vector sequences to match the output vector with the existing vectors in the code (see Figure 6.8(d)). Note that the sequence of the scalars `rC00`, `rC10`, `rC20` and `rC30` inside the vector in Figure 6.8(c) dictates the order of four input vectors of VVRSUM in Figure 6.8(d).

In Algorithm 6.1, we show how FKO vectorizes single basic block in our SLP extension. We skip the implementation details for each steps of our algorithm here, since we have already discussed them in Section 6.1.1. Moreover, the implementations of `FindAdjRef()`, `ExtendPacklist()` and `Schedule()` routines are similar to [31]. We have three types of basic blocks in our simple loopnests: loop block, preheader and posttail. For the loop block, both the initial pack list and the input vector list are empty. We create seed packs based on the adjacent memory access in line 8 (as we already discussed in Section 6.1.1). In the case of the posttail, we delete the reduction codes (if they exist) (lines 10-12). If the input vector list is non-empty (in case of the preheader and the posttail), we attempt to create packs based on the input vectors from the basic block (line 14). If no packs can be formed in this way, we use adjacent memory to form seed packs (line 16). We then sort the seed packs (line 20) based on the sorting criteria passed in as an argument. In line 22, we extend the packs from the sorted seed packs (as we discussed in Section 6.1.1). The scheduling of statements and the actual vectorization are done in line 24. We add VVRSUM code as the last step (if needed).

In Algorithm 6.2, we show briefly how our recursive SLP for loop-nests works. We recur down to the innermost loop and apply our single basic block SLP vectorization on the loop-block (line 5) which returns the created vector list V_o. We then apply single basic block SLP on duplicated preheader (PreBlk) and posttail (PostBlk) block of the innermost loop (line 17 and 18). If the vectorizations are consistent throughout the loop, we update the original preheader and posttail with the vectorized codes (line 24 and 31). If they are not consistent, we gather all live-in (at the entry of the loop) vectors at the end of preheader and

Algorithm 6.1: SLP Vectroization for Single Basic Block

```
(1)  /*INPUT: Basic Block B, Vector-list Vi, sorting_criteria for packs*/
(2)  /*OUTPUT: SIMD vector-list Vo if B successfully vectorized*/
(3)  funct DoSingleBlockSLP (B, Vi, sorting_criteria)
(4)    /*init packset with empty set*/
(5)    P := ∅;
(6)    /*step1: create seed packs*/
(7)    if (V_i = ∅)
(8)      then P := FindAdjRef(B, P, vlen);
(9)       else
(10)          if (isPosttail(B) ∧ isVvrsumNeeded(B, V_i))
(11)            B := DelReductCode(B, V_i);
(12)            isvvrsum := true;
(13)          fi
(14)          P := FindPackFromVlist(V_i);
(15)          if (P := ∅)
(16)            P := FindAdjRef(B, P, vlen);
(17)          fi
(18)     fi
(19)     /*sort seed packs based on criteria provided*/
(20)     P := SortPacks(P, sorting_criteria);
(21)     /*Step2: extend packs from seed packs*/
(22)     P := ExtendPacklist(B, P);
(23)     /*Step3: schedule statements and emit vector statements*/
(24)     [B, V_o] := Schedule(B, P);
(25)     /*add vvrsum code if applicable*/
(26)     if (isvvrsum = ture)
(27)       B := AddVVRSUM(V_i, V_o, B);
(28)     fi
(29)     return(V_o);
```

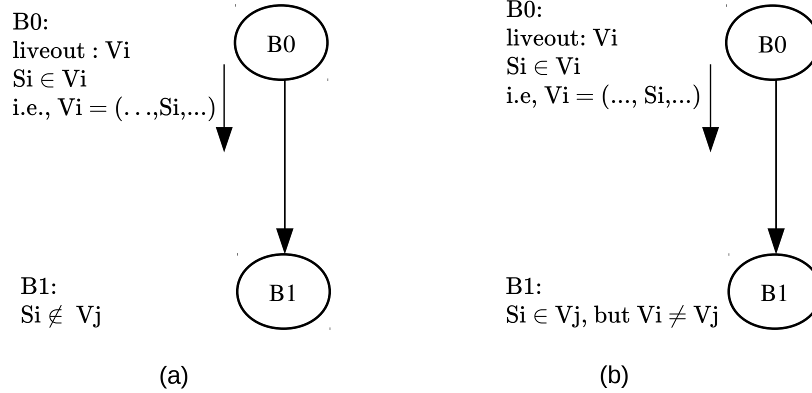


Figure 6.9: Inconsistent vectorization: (a) variable Si is used as scalar in successor block (b) variable Si is element of different vector in successor block

scatter live-out (at the exit of the loop) vectors to scalars at the beginning of the posttail (line 26 and 33). Vectorization are inconsistent when we have a mismatch in vectors between two adjacent blocks. Figure 6.9 shows two cases where the vectorization of a block $B1$ can be inconsistent with its predecessor $B0$. In Figure 6.9(a), Vi is liveout from the block $B0$ and the scalar variable Si is element of the vector. However, Si in block $B1$ is not part of any vector. Therefore, without scattering the vector Vi to Si at the beginning of the block $B1$, the vectorization is not consistent. In Figure 6.9(b), Si is used to form the vector Vj but it is not consistent with the Vi at its predecessor block $B0$. Therefore, the vectorization of $B1$ is inconsistent. This inconsistency can be resolved by shuffling the elements of the vector, but this operation is expensive on some architectures. We therefore avoid this shuffling of vector elements in our implementation and we discard vector codes of preheader and posttail if that ever happens. However, it is very unlikely to happen in well written HPC kernels. Note that even if we suspend vectorization on this block for inconsistency, it may be possible to vectorize other blocks of outer loops in our method. Note that vectorization can be consistent even if SLP fails on either the preheader or the posttail. If the vectorization is successful for this loop, we add all the vectors generated during the vectorization process to the output list and return it to upper level of loop. This process repeats until we we have vectorized all nested loops within this loop nest.

Algorithm 6.2: Loopnests SLP Vectorization

```
(1)  funct DoLoopNestsVec(LOOP, sorting_criteria)
(2)    /*exit condition: reach innermost loop*/
(3)    if (LOOP_i = LOOP_0)
(4)      then
(5)          V_o := DoSingleBlkSLP(LOOP_0.blk, NULL, sorting_criteria);
(6)          returnV_o;
(7)    fi
(8)    /*recursion on next deep level loop*/
(9)    V_o1 := DoLoopNestsVec(LOOP_{i-1}, sorting_criteria);
(10)   /*copy scalar prehead and posttail to apply SLP on*/
(11)   PreBlk := Clone(LOOP_{i-1}.Prehead);
(12)   PostBlk := Clone(LOOP_{i-1}.Posttail);
(13)   /*find live-in,live-out vectors*/
(14)   V_in := FindLiveInVector(V_o1, LOOP_{i-1});
(15)   V_out := FindLiveOutVector(V_o1, LOOP_{i-1});
(16)   /*attempt SLP on scratched preheader and posttail*/
(17)   V_o2 := DoSingleBlkSLP(PreBlk, V_in, sorting_criteria);
(18)   V_o3 := DoSingleBlkSLP(PostBlk, V_out, sorting_criteria);
(19)   /*check vector consistency of loop, prehead and posttail*/
(20)   if (isConsistent(LOOP_i, PreBlk, PostBlk))
(21)     then
(22)         /*if SLP is successful in PreBlk, update preheader with vec code*/
(23)         if (V_o2 ≠ ∅)
(24)           then LOOP_{i-1}.Prehead := PreBlk;
(25)            else
(26)                LOOP_{i-1}.Prehead := AddVectorGather(LOOP_{i-1}.Prehead, V_in);
(27)         fi
(28)         /*if SLP is successful in PostBlk, update posttail with vec code; */
(29)         if (V_o3 ≠ ∅)
(30)           then
(31)                LOOP_{i-1}.Posttail := PostBlk;
(32)            else
(33)                LOOP_{i-1}.Posttail := AddVectorScatter(LOOP_{i-1}.Posttail, Vout);
(34)         fi
(35)         V_o := Combine(V_o1, V_o2, V_o3);
(36)         return(V_o);
(37)         /*inconsistent, discard vector code, add scatter/gather code*/
(38)     else
(39)         LOOP_{i-1}.Prehead := AddVectorGather(LOOP_{i-1}.Prehead, V_in);
(40)         LOOP_{i-1}.Posttail := AddVectorScatter(LOOP_{i-1}.Posttail, V_out);
(41)         return(NULL);
(42)   fi
```

6.1.4 Restriction and Correctness

The original SLP vectorization [31] is supposed to vectorize a single basic block. So, vector-gather operations are used at the beginning to create vectors which consist of the scalars that are live-in at the start of the block. We do not want to keep scatter/gather code inside the loop, so we hoist the gather code up to preheader and push the scatter code down to the posttail and out of the loop. Not only do we hoist and push them out, but also try to initiate SLP with the vectors to vectorize preheaders and posttails so that we can continue to hoist and push them towards outer loops and even eliminate them (if we reach to their definitions and vectorization is successful). We therefore push the vectorization both ways (up and down) from innermost loop (unlike bottom-up approach from memory-store found in LLVM). This is key to our success in achieving very high performance in our studied *gemm*μ kernels. To make the process simpler, we need a single point of entry and exit to and from the loop. Pre-header and posttail serve this purpose. Note that the idea of a single entry to loop (preheader) is already well-known for natural loop (compiler may create preheader to achieve that)[1]. To expand our vectorization from innermost loop to outer-loop, we assume single exit of loop (this is true for all countable loops) as well. We therefore maintain a strict loop structure which we call simple loop-nests, defined in Definition 1 where we don't allow any branch other than the branch needed for the back-edge of the loop. Our SLP vectorization inside a single block is similar to the paper [31]. FKO performs dependence analysis to validate the scheduling of the scalar statements to emit vector code. Therefore, as long as the scheduling of statements does not violate the dependence of original program, the vectorization will not produce any wrong code. Since in our loop structure, there is only one entry to and exit from the loop, hoisting and pushing the live-in and live-out vectors out of the loop also produces valid code. We test the consistency of vectorization among blocks along all potential edges (preheader to header, tail to posttail and posttail to preheader). By consistency we mean: no scalar which is an element of live-out vectors is used as a scalar in

a successor block. If this scalar is used as an element of any vector in successor block, both vectors in predecessor and successor must be same. If our consistency test fails, we avoid vectorization in preheader and posttail and added vector-gather at the end of preheader and vector-scatter code at the beginning of posttail. However, the outer level of loops (beyond this preheader and posttail) may still be vectorized if the vectorization is proved valid for them.

6.1.5 Hybrid Approach: Innermost by Loop Vectorization and Outer by SLP

In our loop-nest vectorization, other vectorization methods can be used to vectorize the innermost loop. For example, we can apply FKO's no hazard loop vectorization in the innermost loop but extend vectorization by SLP outwards from preheader and posttail of the innermost loop. We can vectorize the loopnests of the L2BLAS (e.g., GEMVT) kernels with this hybrid approach. Speculative vectorization can also be used to vectorize the innermost loop in this approach. However, we have not added speculative vectorization in our current implementation of this hybrid approach.

6.2 Experiments and Results

To validate the effectiveness of our auto-vectorization technique, we have performed three different experiments. In the first two experiments (described in Section 6.2.1 and Section 6.2.2), we compare the effectiveness of the autovectorization methods of different compilers on *gemmμ* kernels. The third experiment, which is described in Section 6.2.3, compares the results of our autovectorization technique with ATLAS's existing generated SIMD intrinsic kernels and hand tuned kernels on four different machines.

6.2.1 Tuning *gemmμ* on ATLAS with Autovectorization of Different Compilers

To fairly test auto-vectorization capabilities, we produced a code generator that could generate scalar code (ATLAS's original code generator is manually vectorized using intrinsics). We then ran the full tuning search for each compiler to find the highest performing autovectorized *gemmμ* each compiler is capable of producing. Table 6.1 summarizes the tuning parameters

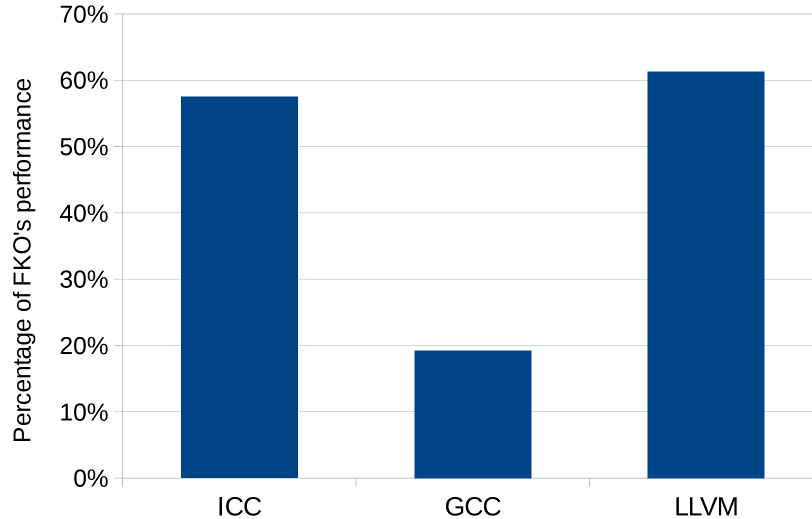


Figure 6.10: Best-case autovectorization performance of various compilers as a percentage of the performance FKO achieves for an Intel Haswel machine

found by the search for each compiler on Intel Core i5-4670 machine. Figure 6.10 then shows the performance these best-case kernels achieved as a percentage of FKO's. As evident from Figure 6.10, LLVM and ICC can manage only around 60% of FKO's performance on this machine, while GCC achieves less than 20%.

6.2.2 Comparing Autovectorization of Different Compilers for Specific *gemmµ* Kernels

To investigate the failures of autovectorization of the industry standard compilers from our previous experiment, we selected three kernels, timed them and then analyzed the output assembly produced by each compiler (see Appendix B for example *gemmµ* listings). Figure 6.11 shows the performance of different compilers' autovectorization in terms of percentage of FKO's performance for these selected kernels. FKO vectorized all the basic blocks of their loopnests, while other compilers failed to vectorize them effectively. The summary of

Table 6.1: ATLAS's tuning results for *gemmµ* using autovectorization of different compilers

Compiler	Kernel selected	Block Factor
FKO	dmvec12x4x1	168x168x179
ICC	dmvec4x4x1	304x304x304
GCC	dmvec12x1x1	252x252x261
LLVM	dmvec12x4x1	168x168x179

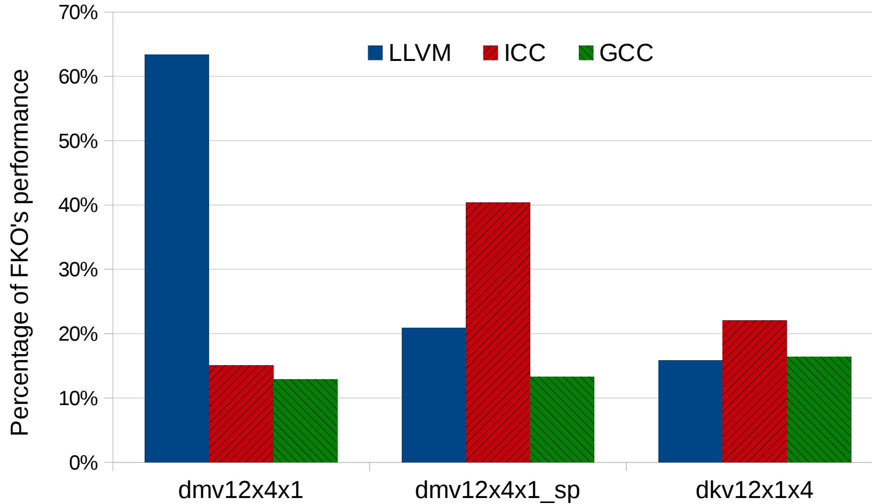


Figure 6.11: Autovectorizaton performance of LLVM (solid blue), ICC (right upward diagonal hashed red) and GCC (right downward diagonal hashed green) as a percentage of the performance FKO achieves for three specific *gemmμ* kernels on Intel Haswel machine.

our findings are as follows:

1. **dmv12x4x1**: This MVEC kernel with $u_m = 12$, $u_n = 4$ and $u_k = 1$ was selected as the best kernel on the machine when FKO and LLVM were used (a similar MVEC kernel was selected for hand tuned and the intrinsic generator). LLVM vectorized all operations of the whole loopnests of this kernel. However, it generated vector-shuffle instructions and spilled registers inside the innermost loop which caused its performance loss. ICC on the other hand only vectorized the innermost loop, treating the memory access as strided and therefore generated a large number of shuffle instructions inside the innermost loop, which results in substantial performance loss compared to FKO or LLVM. GCC only vectorized the posttail and failed to vectorize any other basic blocks in the loopnests.

2. **dmv12x4x1_sp**: Analysis of the prior (best case) kernel indicated that part of ICC's problem with it was due to live-in scalar variables. To aid the compiler, we modified the kernel by jamming all the loads of pA and minimizing the live-in variables at the entry of the innermost loop. ICC could now vectorize the preheader along with

Table 6.2: Machine specification

Machine	Processor	Speed	FPU support
Corei34 (skylake)	Intel Corei 7-6700K CPU	4.0GHz	AVX2
corei324 (haswell)	Intel Xeon E5-2670	2.3 GHZ	AVX2
corei212 (ivy bridge)	Intel Xeon E5-2620	2.1 GHz	AVX
Opt32 (Opteron)	AMD Opteron 6128	2.0 GHz	SSE4.1

the innermost loop, but generated extra reduction codes on posttail (due to failing to apply SLP). The icc-improving modification strongly reduced LLVM's performance by removing its ability to vectorize the inner-loop predecessor blocks. As before, GCC failed to vectorize any block other than posttail as well.

3. **dkv12x1x4**: This KVEC kernel with $u_m = 12$, $u_n = 1$ and $u_k = 4$ was the best KVEC kernels selected when FKO was used (the same tuning parameters were used in the intrinsic generator when tuning KVEC kernels). FKO not only vectorized all the blocks in loop-nests, but also applied VVRSUM codes to parallelize the reduction codes. All other compilers only managed to vectorize the innermost loop.

6.2.3 Comparing Autovectorization of FKO with SIMD Intrinsic and Handtuned Kernels in ATLAS

As our final experiment, we compared FKO's autovectorization with ATLAS's generated SIMD intrinsic and hand tuned kernels on different machines. Hand tuned kernels performed the best since they are written in assembly exploiting architecture-specific optimizations. Figure 6.12 shows the peak performance of FKO's autovectorization (using modified generator) and SIMD intrinsic (using ATLAS's intrinsic generator) as a percentage of performance achieved by hand tuned kernels after running the full search. Figure 6.12 shows best-case performance between FKO's autovectorization and SIMD intrinsic as the percentage of the performance of hand tuned codes on different machines. Table 6.2 summarizes the machine specifications used in this experiment.

The performance of FKO's autovectorization is comparable with the ATLAS's SIMD intrinsic generator. FKO even performed better on corei34 where GCC spilled registers inside

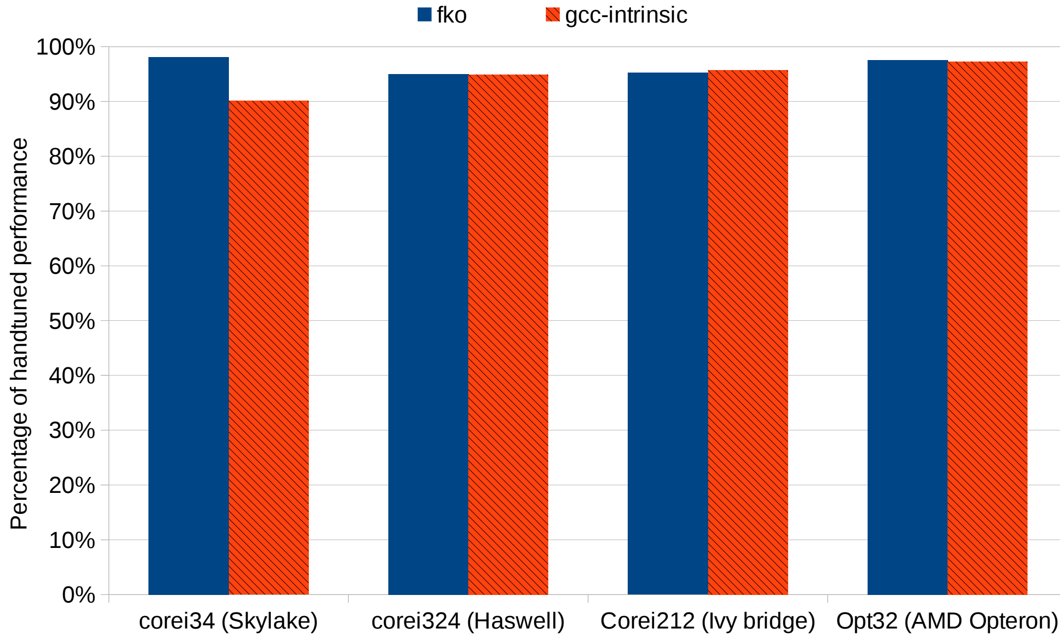


Figure 6.12: Best-case performance between FKO's autovectorization (solid blue) and GCC SIMD-intrinsic (hashed red) as the percentage of performance hand tuned codes achieve in ATLAS on different machines

innermost loop when compiling the intrinsic code. Hence the best kernel for that machine was not chosen in this case. The fact that our autovectorization is always competitive shows that our modification of SLP is extremely effective.

6.3 Related Work

Larsen and Amarasinghe [31] were the first to present super-word level parallelism (SLP) vectorization which dealt with straight line of code in basic block. Their algorithm is simple but effective in vectorizing basic blocks. This technique has been adapted by most of the industry compilers (e.g., GCC, ICC, LLVM) [50, 38, 39]. Shin, Hall and Chame [57] extends SLP to vectorize blocks with dynamic control flow. Rosen, Nuzman and Zaks [39] showed how SLP can be used to vectorize innermost loop after using loop unrolling in their "Loop-ware" SLP vectorization method. To minimize the difficulties to find isomorphic statements in packs, Porpodas, Magni and Jones [43] proposed "PSLP" which padded redundant instructions to transform non-isomorphic sequence into isomorphic ones. Gao et al. [26] proposed "Insufficient Vectorization" where they vectorized code with partial use of vector register when the

inherent parallelization of the code is poor. Porpodas and Jones [42] showed how limiting SLP vectorization by pruning the dependence graph can increase the overall performance since it may reduce the penalty of vector shuffle instructions for the scatter/gather operation SLP introduced. None of these methods solve the specific problem of the whole loopnests vectorization for our HPC kernels.

6.4 Conclusions and Future Work

This chapter presents a new approach to apply autovectorization on loopnests which extends the existing SLP vectorization beyond a single basic block and the innermost loop by initializing the packs of outer loops with the live-in and live-out vectors created in inner loops and combining the parallelization of reduction codes with a special sequence codes (VVR-SUM). We have implemented this technique in our compilation framework and interfaced it with ATLAS. Our technique can effectively vectorize the complete loopnests of all the $gemm\mu$ kernels in ATLAS and can achieve up to 98% performance of ATLAS's hand tuned kernels. It significantly outperforms the autovectorizations of industry compilers for those kernels. However, our extended SLP vectorization works well on simple loopnest defined in this chapter where the loopnests do not have any branch other than the loop-branch and this restriction suits well for our studied HPC kernels. Note that our technique can still vectorize the innermost loop (and partial loopnests which follow the definition) even if the whole nested-loop is not a simple loopnest. Moreover, we have combined this technique with other innermost vectorization technique (e.g., traditional loop vectorization) to vectorize the L2BLAS kernels. We believe it can also be combined with our speculative vectorization which works in presence of conditional branches in loops. We have not found any HPC kernels requiring this combinations yet. When we do, we will explore how we can combine speculative vectorization and SLP to vectorize more complex loopnests.

CHAPTER 7

REPRESENTATION OF TWO DIMENSIONAL ARRAY IN FKO

FKO supports two dimensional column major arrays. In column major, the elements of a column are consecutive and the elements of a row are strided. We refer to the stride between elements in a row as the "leading dimension of the array" (`lda`). In Figure 7.1, we consider a matrix with number of rows $M = 5$, number of columns $N = 3$ and the leading dimension `lda`=6 and it shows both the (a) logical storage and (b) physical memory for that matrix in column major format of array storage. Note that `lda` $>= M$ (i.e., the number of rows M in matrix can be different than the actual leading dimension of the array). Also note that the elements of the first column (e.g., (0,0), (1,0), (2,0), etc) in Figure 7.1(a) are actually stored in physical memory consecutively in Figure 7.1(b) and the elements of the first row (e.g., (0,0), (0,1), (0,2)) of the array in Figure 7.1(a) are stored `lda` (six elements) apart (not M) in actual physical memory in Figure 7.1(b).

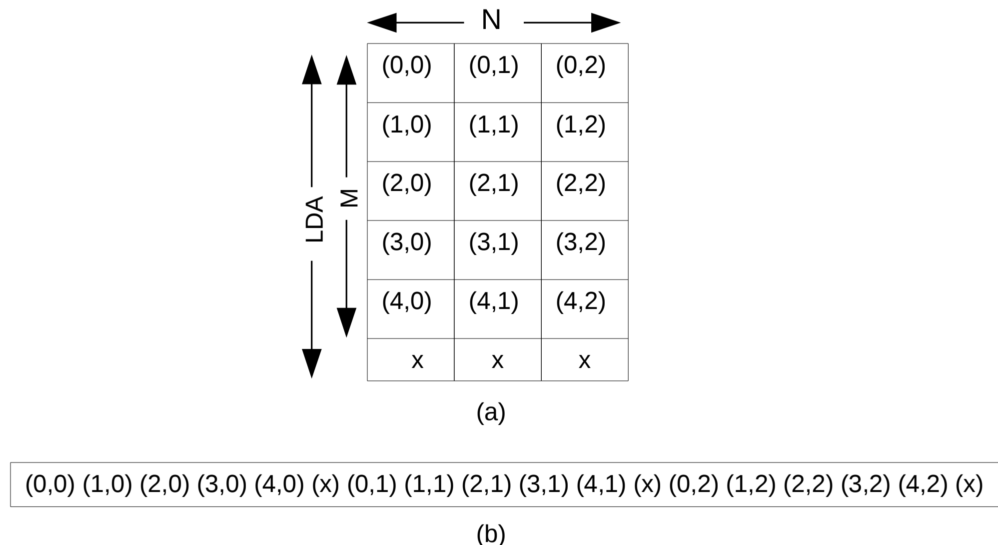


Figure 7.1: Column-major storage of a two-dimensional array with M=5, N=3 and `lda`=6 : (a) Logical storage (b) Physical memory

```
1          (a) DECLARATION
2          ---------------
3             TYPE [stride-between-rows][*] :: array-name;
4             UNROLL_ARRAY :: array-name(row-unroll-factor,column-unroll-factor);
5
6          (b) EXAMPLE
7          -----------
8             DOUBLE_ARRAY [lda][*] :: A;
9             UNROLL_ARRAY :: A(*,6);
```

Figure 7.2: Declaration and example of pointer to two dimensional array in FKO

7.1 Declaration of Pointer to Two Dimensional Array in FKO

In FKO, a pointer can be declared as a pointer to a two dimensional array. This declaration has two parts: dimension information, and unroll information. As dimension information, we need to specify the stride between rows (mandatory run- or compile-time constant LDA) and number of rows (optional) in the array. Unroll information is used to provide FKO a heads-up about the maximum index of column used inside the whole routine to access any element of the array, so that FKO can pre-compute all the addresses of column without analyzing the code. Note that we can provide unroll factor for rows but it is optional and LDA can be a constant or a variable but the max unroll factor must be a constant. Figure 7.2 shows the basic structure and an example of a pointer to two dimensional array. In the example of Figure 7.2(b), we have declared a double array, A, which has `lda` stride between the elements in a row and a column unroll factor of six (`lda` would typically be an integer parameter to the routine). Max unroll 6 implies a column index between 0 and 5. Figure 7.3 shows an example of a routine using a two dimensional array in FKO's input language. This is a L2BLAS kernel (GEMVT) where the unroll and jam[4] optimization has been applied. The unroll factor of the outer loop-unrolling is six. Inside the inner loop, we are accessing six columns in a single iteration. The array A is declared with the row stride of `lda`. So, the address of any element of array, A[i][j], can arithmetically be computed by

```
A + lda*j + i
```

To avoid costly arithmetic (multiplications) inside the innermost loop, FKO pre-computes the address of each column and stores all (or, some) of them in internal variables.

```
1     ROUTINE ATL_UGEMV;
2        PARAMS :: M, N, A, lda, X, Y;
3        INT :: M, N, lda;
4        DOUBLE_PTR :: X, Y;
5        DOUBLE_ARRAY [lda][*] :: A;
6        UNROLL_ARRAY :: A(*,6);
7     ROUT_LOCALS
8        INT :: i, j, ldam;
9        DOUBLE :: y0, y1, y2, y3, y4, y5, a0, a1, a2, a3, a4, a5, a6, x0;
10    ROUT_BEGIN
11       ldam = lda * 6;
12       ldam = ldam - M;
13       j = N;
14       NLOOP:
15          y0 = Y[0];
16          y1 = Y[1];
17          y2 = Y[2];
18          y3 = Y[3];
19          y4 = Y[4];
20          y5 = Y[5];
21
22          LOOP i = 0, M
23          MUTUALLY_ALIGNED(32) :: *;
24          LOOP_BODY
25             x0 = X[0];
26             a0 = A[0][0];
27             y0 += a0 * x0;
28             a1 = A[0][1];
29             y1 += a1 * x0;
30             a2 = A[0][2];
31             y2 += a2 * x0;
32             a3 = A[0][3];
33             y3 += a3 * x0;
34             a4 = A[0][4];
35             y4 += a4 * x0;
36             a5 = A[0][5];
37             y5 += a5 * x0;
38             A += 1;
39             X += 1;
40          LOOP_END
41          Y[0] = y0;
42          Y[1] = y1;
43          Y[2] = y2;
44          Y[3] = y3;
45          Y[4] = y4;
46          Y[5] = y5;
47          A += ldam;
48          Y += 6;
49          X = X-M;
50          j = j - 6;
51          IF (j > 0) GOTO NLOOP;
52    ROUT_END
```

Figure 7.3: Example of two dimensional array in HIL code for DGEMVT kernel with max unroll-factor six

In the following two sections, we will first discuss how FKO translates the memory address of a two dimensional array in general for all architectures and then, after discussing the powerful addressing mode of the x86, we will discuss how FKO exploits that to optimize the performance of memory addressing for such array.

7.1.1 Default 2D Array Addressing

When no architecture specific optimization is applied, FKO converts a pointer to a two dimensional array into multiple pointers to the one dimensional columns of the array using the `lda` and the unroll factor specified in the declaration. The `lda` can either be a variable or a constant. Since FKO treats all two dimensional arrays as column major, it creates pointers to point at each unrolled column of the array simply, with the next column pointer initialized by adding `lda` to the prior column's pointer. For a column unroll factor of six, FKO would therefore use six 1D pointers. In lines 12-17 of Figure 7.4, we see how FKO initializes its internal column pointers (e.g.,`_A0`, `_A1`, ..., `_A5`) to point at each column of the array. The access of the first element in the first column, `A[0][0]`, is converted into `_A0[0]`, the first element in the second column, `A[0][1]`, into `_A1[0]` and so on (see lines 33, 35, 37, 39, 41 and 43 of Figure 7.4). The increment of the pointer to the 2D array, A, is therefore converted into increments to six column-pointers (see lines 45-50 and 59 of Figure 7.4). Note that although this approach saves integral arithmetic operations when addressing the array within the loop, and therefore shortens dependence chains, it increases the register pressure inside the loop. We now need five extra registers to hold all the column addresses. At the end of the loop, we also need six addition operations to update the pointers. We will now discuss how we can minimize the register pressure on the x86 using its powerful addressing mode next.

7.1.2 Exploiting x86 Addressing Modes for 2D Arrays

The x86 architecture has a very flexible addressing mode to calculate the effective address:

$$\text{Effective_Address} = \text{Base_Address} + (\text{Index} * \text{Multiplier}) + \text{Constant_offset}$$

```
1        ROUTINE ATL_UGEMV;
2           PARAMS :: M, N, A, lda, X, Y;
3           INT :: M, N, lda;
4           DOUBLE_PTR :: X, Y;
5           DOUBLE_ARRAY [lda][*] :: A;
6           UNROLL_ARRAY :: A(*,6);
7        ROUT_LOCALS
8           INT :: i, j, ldam;
9           DOUBLE :: y0, y1, y2, y3, y4, y5, a0, a1, a2, a3, a4, a5, a6, x0;
10       ROUT_BEGIN
11          // compiler's internal ptr arith
12          _A0 = A;
13          _A1 = _A0 + lda;
14          _A2 = _A1 + lda;
15          _A3 = _A2 + lda;
16          _A4 = _A3 + lda;
17          _A5 = _A4 + lda;
18          // end of compiler's internal ptr arith
19          ldam = lda * 6;
20          ldam = ldam - M;
21          j = N;
22          NLOOP:
23             y0 = Y[0];
24             y1 = Y[1];
25             y2 = Y[2];
26             y3 = Y[3];
27             y4 = Y[4];
28             y5 = Y[5];
29             LOOP i = 0, M
30             MUTUALLY_ALIGNED(32) :: *;
31             LOOP_BODY
32                x0 = X[0];
33                a0 = _A0[0];
34                y0 += a0 * x0;
35                a1 = _A1[0];
36                y1 += a1 * x0;
37                a2 = _A2[0];
38                y2 += a2 * x0;
39                a3 = _A3[0];
40                y3 += a3 * x0;
41                a4 = _A4[0];
42                y4 += a4 * x0;
43                a5 = _A5[0];
44                y5 += a5 * x0;
45                _A0 += 1;
46                _A1 += 1;
47                _A2 += 1;
48                _A3 += 1;
49                _A4 += 1;
50                _A5 += 1;
51                X += 1;
52             LOOP_END
53             Y[0] = y0;
54             Y[1] = y1;
55             Y[2] = y2;
56             Y[3] = y3;
57             Y[4] = y4;
58             Y[5] = y5;
59             _A0 += ldam; _A1 += ldam; _A2 += ldam; _A3 += ldam; _A4 += ldam; _A5 += ldam;
60             Y += 6;
61             X = X-M;
62             j = j - 6;
63             IF (j > 0) GOTO NLOOP;
64       ROUT_END
```

Figure 7.4: Pseudocode for the address-translation in general approach done by FKO for DGEMVT kernel shown in Figure 7.3

This addressing mode can use two registers (base and index), one multiplier of value 1, 2, 4, or 8 and a constant offset or displacement. In 2-D array indexing, the base register holds the pointer, while the index register holds multiples of `lda`. We can then calculate column addresses with fewer registers and pointer updates. We assume that the row indices are constant, so they can be encoded in to the constant part of the addressing mode. Figure 7.5 illustrates how FKO minimizes the number of column pointers and arithmetic operations by exploiting this addressing mode. Note that this figure shows pseudocode: Since FKO generates assembly, it calculates addresses in bytes, not elements of array as shown here, but the idea is the same. More generally, Table 7.1 shows the addressing mode arithmetic required to compute all column addresses up to a max unroll of 16. Each element of the row with MUR (max unroll) of 6 shows the calculation used to find the start of the given column index shown along the top row. Suppose we want to compute the address for `A[1][4]` assuming unroll factor six. In our default mode, this would be `A4[1]`, but as shown in Table 7.1, we will use `A2+lda*2+1`, which in assembly form looks like `4(A2, lda,2)` (assuming `A2` and `lda` were actually registers, and `A` is of type `float`). Figure 7.5 shows this optimization applied in FKO's input language (HIL) using a max unroll factor of six. Based on the sixth of row of the Table 7.1, we need one column pointer ($A2$) and three additional variables/registers to save `lda`, (`-1*lda`) and (`3*lda`). In lines 12-14 of Figure 7.5, FKO pre-computes those values into the internal variables (e.g., `_A2`, `_lda_n` and `_lda_3`). Note that `_A2` stores the address of the third column. To compute the column addresses of indices 0-5, FKO uses the following calculations to compute each column address:

$Column1 : A2 + (-lda) * 2$

$Column2 : A2 + (-lda) * 1$

$Column3 : A2$

$Column4 : A3 = A2 + lda * 1$

$Column5 : A4 = A2 + lda * 2$

```
1   ROUTINE ATL_UGEMV;
2      PARAMS :: M, N, A, lda, X, Y;
3      INT :: M, N, lda;
4      DOUBLE_PTR :: X, Y;
5      DOUBLE_ARRAY [lda][*] :: A;
6      UNROLL_ARRAY :: A(*,6);
7   ROUT_LOCALS
8      INT :: i, j, ldam;
9      DOUBLE :: y0, y1, y2, y3, y4, y5, a0, a1, a2, a3, a4, a5, a6, x0;
10  ROUT_BEGIN
11     // compiler's internal ptr arithmatic
12     _A2 = A + 2*lda;
13     _lda_n = - lda;
14     _lda_3 = 3 * lda;
15     // end of compiler's internal ptr arithmatic
16     ldam = lda * 6;
17     ldam = ldam - M;
18     j = N;
19     NLOOP:
20        y0 = Y[0];
21        y1 = Y[1];
22        y2 = Y[2];
23        y3 = Y[3];
24        y4 = Y[4];
25        y5 = Y[5];
26
27        LOOP i = 0, M
28        MUTUALLY_ALIGNED(32) :: *;
29        LOOP_BODY
30           x0 = X[0];
31           a0 = (_A2,_lda_n, 2)[0];   // A0 = A2 - 2*lda
32           y0 += a0 * x0;
33           a1 = (_A2, _lda_n)[0];     // A1 = A2 - lda
34           y1 += a1 * x0;
35           a2 = _A2[0];               // A2
36           y2 += a2 * x0;
37           a3 = (_A2,lda)[0];         // A3 = A2 + lda
38           y3 += a3 * x0;
39           a4 = (_A2,lda,2)[0];       // A4 = A2 + 2*lda
40           y4 += a4 * x0;
41           a5 = (_A2, _lda_3)[0];     // A5 = A2 + 3*lda
42           y5 += a5 * x0;
43           _A2 += 1;
44           X += 1;
45        LOOP_END
46        Y[0] = y0;
47        Y[1] = y1;
48        Y[2] = y2;
49        Y[3] = y3;
50        Y[4] = y4;
51        Y[5] = y5;
52        _A2 += ldam;
53        Y += 6;
54        X = X-M;
55        j = j - 6;
56        IF (j > 0) GOTO NLOOP;
57  ROUT_END
```

Figure 7.5: Pseudocode for the optimized address-translation done by FKO for DGEMVT kernel shown in Figure 7.3

$Column6 : A5 = A2 + (3 * lda) * 1$

Figure 7.5 uses such calculation in lines 31, 33, 35, 37, 39 and 41). In this way, FKO can access those six columns with only four registers. Another advantage of this technique is that we need only one addition operation to update the pointer($_A2$) at the end of the loop (see line 43 of Figure 7.5). In contrast, the general approach (discussed in previous section) uses six registers and needs six addition operations to update each column pointer.

Table 7.2 summarizes the register usage and arithmetic operations needed for pointer updates for each max unroll factor. It also provides information on how much we can save in register usage and pointer updates over the general approach we discussed before. For example, this technique does not use any less registers until we get to an unroll factor of 4. Note, however, that it starts saving updates at a max unroll of only 2: this type of discrepancy is because this optimization replaces loop-variable pointers with loop-invariant indices, enabling FKO to limit updates even when the register pressure is not reduced. As the max unroll is increased, we see that both the register and update savings increase as well.

7.2 Experiments and Results

We have performed an experiment to validate the effectiveness of our 2D array addressing optimization. The kernels we chose for this experiment were the double precision level-2 GEMVT kernels with different unroll and jam factors. We timed these kernels using ATLAS's timing framework on Intel Haswell machine (Intel Core i5-4670 processor). We used in-cache data for the timing. We enabled FKO's no-hazard loop vectorization and didn't perform any other tuning for those kernels. Table 7.2 shows the benefit of using optimized 2D array addressing over the general approach for these kernels. The greater unroll factor we use, the more benefit we get in terms of the savings of registers and arithmetic operations needed for updating the pointers. However, these kernels are floating point computation heavy and the register pressure for floating points increases with the larger unroll factor. We therefore did

Table 7.1: Rows of the table show the maximum unroll factor used in the loop, while the columns show the column index of the 2-D array (which start from zero). The cells then show the indexing computation required to go to that column, with indices beyond the max unroll set to n/a. All multiplications and additions are done using the x86 addressing modes, while subtractions require additional registers to hold the negative values, as does L3, which holds 3*L. Note that for max unroll $\geq$ 9 you must consult both subtables to get all valid indices, which have been split to fit the page.

IDX/ MUR	0	1	2	3	4	5	6	7
1	A0	n/a	n/a	n/a	n/a	n/a	n/a	n/a
2	A0	A0+L	n/a	n/a	n/a	n/a	n/a	n/a
3	A2-L*2	A2-L	A2	n/a	n/a	n/a	n/a	n/a
4	A2-L*2	A2-L	A2	A2+L	n/a	n/a	n/a	n/a
5	A2-L*2	A2-L	A2	A2+L	A2+L*2	n/a	n/a	n/a
6	A2-L*2	A2-L	A2	A2+L	A2+L*2	A2+L3	n/a	n/a
7	A2-L*2	A2-L	A2	A2+L	A2+L*2	A2+L3	A2+L*4	n/a
8	A2-L*2	A2-L	A2	A2+L	A2+L*2	A7-L*2	A7-L	A7
9	A2-L*2	A2-L	A2	A2+L	A2+L*2	A7-L*2	A7-L	A7
10	A2-L*2	A2-L	A2	A2+L	A2+L*2	A7-L*2	A7-L	A7
11	A2-L*2	A2-L	A2	A2+L	A2+L*2	A7-L*2	A7-L	A7
12	A2-L*2	A2-L	A2	A2+L	A2+L*2	A7-L*2	A7-L	A7
13	A2-L*2	A2-L	A2	A2+L	A2+L*2	A9-L*4	A2+L*4	A9-L*2
14	A2-L*2	A2-L	A2	A2+L	A2+L*2	A9-L*4	A2+L*4	A9-L*2
15	A2-L*2	A2-L	A2	A2+L	A2+L*2	A7-L*2	A7-L	A7
16	A2-L*2	A2-L	A2	A2+L	A2+L*2	A7-L*2	A7-L	A7

IDX/ MUR	8	9	10	11	12	13	14	15
9	A7+L	n/a	n/a	n/a	n/a	n/a	n/a	n/a
10	A7+L	A2+L*2	n/a	n/a	n/a	n/a	n/a	n/a
11	A7+L	A2+L*2	A2+L*8	n/a	n/a	n/a	n/a	n/a
12	A7+L	A7+L*2	A2+L*8	A7+L*4	n/a	n/a	n/a	n/a
13	A9-L*2	A9-L	A9	A9+L	A9+L*2	A9+L3	n/a	n/a
14	A9-L*2	A9-L	A9	A9+L	A9+L*2	A9+L3	A9+L*4	n/a
15	A7+L	A7+L*2	A2+L*8	A7+L*4	A12	A12+L	A12+L*2	n/a
16	A7+L	A7+L*2	A2+L*8	A7+L*4	A12	A12+L	A12+L*2	A7+L*8

not realize the benefit of our optimization in terms of speedup for the larger unroll factors. We got peak performance for this kernel at the unroll-factor=5 and we still managed to achieve 2% speedup using this optimization at this unroll-factor. However, the main advantage of the optimization is that it prevents integer register pressure from inhibiting optimizations that can lead to the best kernel, depending on architecture.

Table 7.2: Registers and updates needed in optimized two dimensional array of DGEMVT code shown in Figure 7.5 and benefit of this translation over general approach shown in Figure 7.4

Unroll Factor	Register needed in optimized addressing			Benefit of optmized addressing over general approach		
	Pointers	Index	Total	Register Saved	Update Saved	Speedup
1	P0		1	0	0	0.99
2	P0	lda	2	0	1	1.01
3	P1	lda, -lda	3	0	2	1.02
4	P1	lda, -lda	3	1	3	1.02
5	P2	lda, -lda	3	2	4	1.02
6	P2	lda, -lda, 3*lda	4	2	5	1.01
7	P2	lda, -lda, 3*lda	4	3	6	1.03
8	P2, P7	lda, -lda	4	4	6	1.02
9	P2, P7	lda, -lda	4	5	7	1.04
10	P2, P7	lda, -lda	4	6	8	1.04
11	P2, P7	lda, -lda	4	7	9	1.04
12	P2, P7	lda, -lda	4	8	10	1.02
13	P2, P9	lda, -lda, 3*lda	5	8	11	1.05
14	P2, P9	lda, -lda, 3*lda	5	9	12	1.06
15	P2, P7, P12	lda, -lda	5	10	12	1.05
16	P2, P7, P12	lda, -lda	5	11	13	1.03

7.3 Conclusions and Future Work

This chapter presents a method to optimize the memory addressing of two dimensional arrays in unrolled and jammed code for the x86 architecture. This addressing optimization is used in handtuned assembly codes inside ATLAS, but we have formalized this technique and implemented it in our compilation framework so that any kernel addressing 2D arrays can reap this benefit without writing in assembly. This optimization minimizes the addressing

computations in the loop, but much more importantly it can significantly reduce integer register pressure for unroll and jammed kernels. This can result in speedup even for the floating point computation heavy kernels (e.g., *gemvt*) as shown in the Table 7.2. Future work includes extending this technique for arrays beyond two dimensions (our present kernel set contains only 1-D or 2-D arrays).

CHAPTER 8
SUMMARY AND CONCLUSIONS

We have picked up the research of the iterative and empirical compilation from where the original iFKO [62] left off. This early effort showed impressive results for all L1BLAS kernels except the two kernels which have conditional branches inside loops. Branches inside loops not only affect performance adversely when misprediction occurs, but also inhibit other compiler optimizations such as SIMD vectorization.

Since SIMD vector units are ubiquitous in modern microprocessors, their effective utilization is critical to attaining high performance. To solve this problem, we have not only implemented the state of the art method to reduce paths for a limited predicate-supported architecture to facilitate vectorization, but also developed a new loop autovectorization technique, speculative vectorization, in our compiler framework. Speculative vectorization is the only known technique that can effectively vectorize and achieve speedup for some important HPC kernels, including one of the two L1BLAS routines (NRM2) that FKO failed to vectorize in [62]. For the other routine that the original work failed to vectorize (IAMAX), we formalized a pre-existing hand-tuning optimization to enable FKO to vectorize it via either speculative vectorization or path reduction.

Further, we have formalized and implemented in FKO special 2-D array indexing support exploiting the x86's rich addressing modes. As far as we are aware we are the first to do so automatically in a compiler. This addressing mode optimization is critical to allowing us to heavily unroll and jam loops without restriction from integer register pressure. Coupled with our vectorization efforts, this allows FKO to tune the L2BLAS to HPC standards.

As for L3BLAS, ATLAS tunes a suite of *gemm*μ kernels. Traditional compilers are unable to effectively autovectorize those microkernels. We therefore have developed an extension of SLP, the state of the art vectorization method for single basic block, to vectorize the complete loopnests of those kernels. Our extended SLP vectorization works well on simple

loopnests (defined in Chapter 6) and can be combined with other innermost loop vectorization technique to extend the vectorization beyond innermost loops, which is critical if the L3BLAS kernels are to be made competitive.

For a few of our surveyed machines, FKO is already within 1-2% of the best hand-tuned *gemm*μ, which is complete success. However, on two other machines our autovectorized *gemm*μ was around 4-5% slower than the best hand-tuned case. That gap is too large for HPC use, so we need some extra autotuning. Initial investigation indicates the main difference is that the hand-tuned code has carefully scheduled prefetch, both within the current block and external to the block. The external-block prefetch is probably best handled at the ATLAS generator level (since it has knowledge of what block will be used next not available to the compiler), but the intra-block prefetch should ideally be tune by a limited iFKO iteration as requested by the GEMM search. This is the first area we will investigate to close the gap between our auto-vectorized kernel and the hand-tuned cases. At that point, a compiler will for the first time be able to tune every kernel used in the ATLAS framework to levels competitive with hand-tuned assembly.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[5] M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *ACM International conference on Supercomputing (ICS)*, June 2008.

[6] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 219–228, Raleigh, NC, USA, 2009. ACM.

[7] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *Int. J. Parallel Program.*, 30(2):65–98, April 2002.

[8] Aart J. C. Bik, Xinmin Tian, and Milind Girkar. Multimedia vectorization of floating-point min/max reductions. *Concurrency - Practice and Experience*, 18(9):997–1007, 2006.

[9] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the ACM SIGARC International Conference on SuperComputing*, Vienna, Austria, July 1997.

[10] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)*, April 2008.

[11] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[12] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to FPGAs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, March 2007.

[13] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, October 2007.

[14] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[15] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 223–234, New York, NY, USA, 2007. ACM.

[16] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[17] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.

[18] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[19] Jialin Dou and Marcelo Cintra. Compiler estimation of load imbalance overhead in speculative parallelization. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 203–214, Washington, DC, USA, 2004. IEEE Computer Society.

[20] R. Dz-ching Ju, K. Nomura, U. Mahadevan, and Le-Chun Wu. A unified compiler framework for control and data speculation. In *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, pages 157 –168, 2000.

[21] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for simd architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 82–93, New York, NY, USA, 2004. ACM.

[22] J. Fisher. Trace scheduling: A technique for global microcode compaction. *Computers, IEEE Transactions on*, C-30(7):478–490, July.

[23] Free Software Foundation. Auto-vectorization in gcc. `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`. [Online; accessed 07-March-2013].

[24] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, 1998.

[25] M. Frigo and S. G. Johnson. The Fastest Fourier Transform in the West. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, 1997.

[26] Wei GAO, Lin HAN, Rongcai ZHAO, Yingying LI, and Jian LIU. Insufficient vectorization: A new method to exploit superword level parallelism. *IEICE Transactions on Information and Systems*, E100.D(1):91–106, 2017.

[27] R. Hanson, F. Krogh, and C. Lawson. A Proposal for Standard Linear Algebra Subprograms. *ACM SIGNUM Newsl.*, 8(16), 1973.

[28] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[29] B. Kågström, P. Ling, and C. van Loan. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. Technical Report UMINF 95-18, Department of Computing Science, Umeå University, 1995. Submitted to ACM TOMS.

[30] R. Karrenberg and S. Hack. Whole-function vectorization. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 141 –150, april 2011.

[31] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM.

[32] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Techniques for increasing and detecting memory alignment. Technical Report MIT-LCS-TM-621, Massachusetts Institute of Technology, 2001.

[33] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[34] Ruby B. Lee. Subword parallelism with max-2. *IEEE Micro*, 16(4):51–59, August 1996.

[35] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 289–299, New York, NY, USA, 2003. ACM.

[36] Dorit Naishlos. Autovectorization in gcc. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2006.

[37] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 132–143, New York, NY, USA, 2006. ACM.

[38] Dorit Nuzman and Ayal Zaks. Autovectorization in gcc–two years later. In *Proceedings of the 2006 GCC Developers Summit*, pages 145–158, 2006.

[39] Dorit Nuzman and Ayal Zaks. Loop-aware slp in gcc. In *Proceedings of the 2007 GCC Developers Summit*, pages 131–142, 2007.

[40] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: revisited for short simd architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM.

[41] Alex Pajuelo, Antonio González, and Mateo Valero. Speculative dynamic vectorization. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 271–280, Washington, DC, USA, 2002. IEEE Computer Society.

[42] Vasileios Porpodas and Timothy M. Jones. Throttling automatic vectorization: When less is more. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 432–444, Washington, DC, USA, 2015. IEEE Computer Society.

[43] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. Pslp: Padded slp automatic vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 190–201, Washington, DC, USA, 2015. IEEE Computer Society.

[44] Dan Quinlan and Chunhua Liao. The rose source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, volume 2011, page 1, 2011.

[45] Dan Quinlan, Haihang You, Qing Yi, Richard Vuduc, and Keith Seymour. Poet: Parameterized optimizations for empirical tuning. *2007 IEEE International Parallel and Distributed Processing Symposium*, 00:447, 2007.

[46] Daniel J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[47] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[48] Lawrence Rauchwerger and David Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. *SIGPLAN Not.*, 30(6):218–232, 1995.

[49] Laurant Rolaz. An implementation of if-conversion using select instructions for machine suif, 2003.

[50] See page for details. Auto-Vectorization in LLVM. http://www.llvm.org/docs/Vectorizers.html.

[51] See page for details. FFTW homepage. http://www.fftw.org/.

[52] See page for details. PLUTO homepage. http://www.pluto-compiler.sourceforge.net.

[53] See page for details. ROSE homepage. http://www.rosecompiler.org.

[54] See page for details. SUIF2 homepage. http://www.suif.stanford.edu/suif/suif2/.

[55] Mark Shabahi. A guide to auto-vectorization with intel c++ compilers. http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers, 2012. [Online; accessed 07-March-2013].

[56] Jaewook Shin. Introducing control flow into vectorized code. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 280–291, Washington, DC, USA, 2007. IEEE Computer Society.

[57] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 165–175, Washington, DC, USA, 2005. IEEE Computer Society.

[58] Yuan Yao Shuai Wei, Rong-Cai Zhao. Loop-nest auto-vectorization based on slp. *Journal of Software*, 2012.

[59] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[60] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *Int. J. Parallel Program.*, 28(4):363–400, August 2000.

[61] Majedul Haque Sujon, R. Clint Whaley, and Qing Yi. Vectorization past dependent branches through speculation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 353–362, Piscataway, NJ, USA, 2013. IEEE Press.

[62] R. Clint Whaley. *Automated Empirical Optimization of High Performance Floating Point Kernels*. PhD thesis, Florida State University, December 2004.

[63] R. Clint Whaley and Anthony M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. Technical Report CS-TR-2008-001, University of Texas at San Antonio, January 2008.

[64] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. http://www.netlib.org/lapack/lawns/lawn131.ps.

[65] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, San Antonio, TX, USA, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category.**
http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps.

[66] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[67] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[68] R. Clint Whaley and Antoine Petitet. Atlas homepage. http://math-atlas.sourceforge.net/, 2011.

[69] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[70] R. Clint Whaley and David B. Whalley. Tuning high performance kernels through empirical compilation. In *The 2005 International Conference on Parallel Processing*, pages 89–98, Oslo, Norway, June 2005.

[71] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The suif compiler system: A parallelizing and optimizing research compiler. Technical report, Stanford, CA, USA, 1994.

[72] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih wei Liao, Chau wen Tseng, Mary Hall, Monica Lam, and John Hennessy. An overview of the suif compiler system. Technical report.

[73] Qing Yi. Poet: A scripting language for applying parameterized source-to-source program transformations. *Softw. Pract. Exper.*, 42(6):675–706, June 2012.

APPENDIX A

ATLAS GEMM MICROKERNELS IN HIL

This appendix provides the HIL implementations (with preprocessor directives) of *gemm*μ kernels discussed in Chapter 6. We show double precision version of MVEC kernel with $u_m = 12$, $u_n = 4$ and $u_k = 1$ in Section A.1 and KVEC kernel with $u_m = 12$, $u_n = 1$ and $u_k = 4$ in Section A.2. Note the conditional branch inside the loopnests of MVEC kernel. Loop unswitching [4] optimization pulls that off the loopnests creating two separate loopnests.

A.1 Double Precision MVEC With $u_m = 12$ $u_n = 4$ $u_k = 1$

```
@ifdef ! ATL_MM_KB
   @define MMKB @0@
@endifdef
@ifdef  ATL_MM_KB
   @define MMKB @1@
@endifdef
@iif MMKB = 0
   @ifdef KB
      @iif KB > 0
         @define ATL_KBCONST @1@
         @define ATL_MM_KB @K@
      @endiif
      @iif KB { 0
        @define ATL_KBCONST @0@
        @define ATL_MM_KB @K@
      @endiif
   @endifdef
   @ifdef ! KB
     @define ATL_KBCONST @0@
     @define ATL_MM_KB @K@
   @endifdef
@endiif
@iif MMKB = 1
  @iif ATL_MM_KB > 0
      @define ATL_KBCONST @1@
  @endiif
  @iif ATL_MM_KB { 0
     @undef ATL_MM_KB
     @define ATL_MM_KB @K@
     @define ATL_KBCONST @0@
  @endiif
@endiif
@ifdef ! BETA1
   @ifdef ! BETA0
       @define ibet @-1@
   @endifdef
@endifdef
@ifdef BETA1
    @define ibet @1@
@endifdef
@ifdef BETA0
   @define ibet @0@
@endifdef
ROUTINE ATL_USERMM;
   PARAMS :: nmus, nnus, K, pA, pB, pC, pAn, pBn, pCn;
   INT :: nmus, nnus, K;
   DOUBLE_PTR :: pA, pB, pC, pAn, pBn, pCn;
//
// Performs a GEMM with M,N,K unrolling (& jam)
//    of (12,4,1).
// Vectorization of VLEN=1 along M dim, vec
//    unroll=(12,4,1).
// You may set compile-time constant K dim by
//    defining ATL_MM_KB.
//
ROUT_LOCALS
   DOUBLE :: rB0, rC0_0, rC1_0, rC2_0, rC3_0,rC4_0,
             rC5_0,  rC6_0,  rC7_0,  rC8_0,  rC9_0,
             rC10_0,  rC11_0,  rC0_1, rC1_1, rC2_1,
             rC3_1,  rC4_1,  rC5_1,  rC6_1,  rC7_1,
             rC8_1,  rC9_1,  rC10_1, rC11_1, rC0_2,
             rC1_2,  rC2_2,  rC3_2,  rC4_2,  rC5_2,
             rC6_2,  rC7_2,  rC8_2,  rC9_2,  rC10_2,
             rC11_2, rC0_3,  rC1_3,  rC2_3,  rC3_3,
             rC4_3,  rC5_3,  rC6_3,  rC7_3, rC8_3,
             rC9_3,  rC10_3, rC11_3, rA0, rA1, rA2,
             rA3,  rA4,  rA5,  rA6,  rA7,  rA8, rA9,
             rA10, rA11;
   DOUBLE_PTR :: pA0, pB0;
   INT :: i, j, k;
   INT :: incAm, incBn;
@iif ibet ! 0
  @declare "   DOUBLE :: " y n ";"
     @iexp j 0 0 +
     @iwhile j < 4
        @iexp i 0 0 +
        @iwhile i < 12
            rC@(i)_@(j)m
           @iexp i @(i) 1 +
        @endiwhile
        @iexp j @(j) 1 +
     @endiwhile
  @enddeclare
@endiif
ROUT_MARKUP
   ALIGNED(32) :: *;
ROUT_BEGIN
   pB0=pB;
   pA0=pA;
@iif ATL_KBCONST = 0
      incAm = K*12;
      incBn = K*4;
@endiif
@iif ATL_KBCONST ! 0
      incAm = @(ATL_MM_KB)*12;
      incBn = @(ATL_MM_KB)*4;
@endiif
   i = nmus;
   MLOOP:
         j =  nnus;
```

```
NLOOP:
   // Peel K=0 iteration to avoid
   // zero of rCxx and extra add
   rA0 = pA[0];
   rA1 = pA[1];
   rA2 = pA[2];
   rA3 = pA[3];
   rA4 = pA[4];
   rA5 = pA[5];
   rA6 = pA[6];
   rA7 = pA[7];
   rA8 = pA[8];
   rA9 = pA[9];
   rA10 = pA[10];
   rA11 = pA[11];
   pA += 12;
   rB0 = pB[0];
   rC0_0 = rA0 * rB0;
   rC1_0 = rA1 * rB0;
   rC2_0 = rA2 * rB0;
   rC3_0 = rA3 * rB0;
   rC4_0 = rA4 * rB0;
   rC5_0 = rA5 * rB0;
   rC6_0 = rA6 * rB0;
   rC7_0 = rA7 * rB0;
   rC8_0 = rA8 * rB0;
   rC9_0 = rA9 * rB0;
   rC10_0 = rA10 * rB0;
   rC11_0 = rA11 * rB0;
   rB0 = pB[1];
   rC0_1 = rA0 * rB0;
   rC1_1 = rA1 * rB0;
   rC2_1 = rA2 * rB0;
   rC3_1 = rA3 * rB0;
   rC4_1 = rA4 * rB0;
   rC5_1 = rA5 * rB0;
   rC6_1 = rA6 * rB0;
   rC7_1 = rA7 * rB0;
   rC8_1 = rA8 * rB0;
   rC9_1 = rA9 * rB0;
   rC10_1 = rA10 * rB0;
   rC11_1 = rA11 * rB0;
   rB0 = pB[2];
   rC0_2 = rA0 * rB0;
   rC1_2 = rA1 * rB0;
   rC2_2 = rA2 * rB0;
   rC3_2 = rA3 * rB0;
   rC4_2 = rA4 * rB0;
   rC5_2 = rA5 * rB0;
   rC6_2 = rA6 * rB0;
   rC7_2 = rA7 * rB0;
   rC8_2 = rA8 * rB0;
   rC9_2 = rA9 * rB0;
   rC10_2 = rA10 * rB0;
   rC11_2 = rA11 * rB0;
   rB0 = pB[3];
   pB += 4;
   rC0_3 = rA0 * rB0;
   rA0 = pA[0];
   rC1_3 = rA1 * rB0;
   rA1 = pA[1];
   rC2_3 = rA2 * rB0;
   rA2 = pA[2];
   rC3_3 = rA3 * rB0;
   rA3 = pA[3];
   rC4_3 = rA4 * rB0;
   rA4 = pA[4];
   rC5_3 = rA5 * rB0;
rA5 = pA[5];
rC6_3 = rA6 * rB0;
rA6 = pA[6];
rC7_3 = rA7 * rB0;
rA7 = pA[7];
rC8_3 = rA8 * rB0;
rA8 = pA[8];
rC9_3 = rA9 * rB0;
rA9 = pA[9];
rC10_3 = rA10 * rB0;
rA10 = pA[10];
rC11_3 = rA11 * rB0;
rA11 = pA[11];
   pA += 12;
IF (K == 1) GOTO KDONE;
LOOP k = 1, @(ATL_MM_KB), 1
LOOP_BODY
   rB0 = pB[0];
   rC0_0 += rA0 * rB0;
   rC1_0 += rA1 * rB0;
   rC2_0 += rA2 * rB0;
   rC3_0 += rA3 * rB0;
   rC4_0 += rA4 * rB0;
   rC5_0 += rA5 * rB0;
   rC6_0 += rA6 * rB0;
   rC7_0 += rA7 * rB0;
   rC8_0 += rA8 * rB0;
   rC9_0 += rA9 * rB0;
   rC10_0 += rA10 * rB0;
   rC11_0 += rA11 * rB0;
   rB0 = pB[1];
   rC0_1 += rA0 * rB0;
   rC1_1 += rA1 * rB0;
   rC2_1 += rA2 * rB0;
   rC3_1 += rA3 * rB0;
   rC4_1 += rA4 * rB0;
   rC5_1 += rA5 * rB0;
   rC6_1 += rA6 * rB0;
   rC7_1 += rA7 * rB0;
   rC8_1 += rA8 * rB0;
   rC9_1 += rA9 * rB0;
   rC10_1 += rA10 * rB0;
   rC11_1 += rA11 * rB0;
   rB0 = pB[2];
   rC0_2 += rA0 * rB0;
   rC1_2 += rA1 * rB0;
   rC2_2 += rA2 * rB0;
   rC3_2 += rA3 * rB0;
   rC4_2 += rA4 * rB0;
   rC5_2 += rA5 * rB0;
   rC6_2 += rA6 * rB0;
   rC7_2 += rA7 * rB0;
   rC8_2 += rA8 * rB0;
   rC9_2 += rA9 * rB0;
   rC10_2 += rA10 * rB0;
   rC11_2 += rA11 * rB0;
   rB0 = pB[3];
   pB += 4;
   rC0_3 += rA0 * rB0;
   rA0 = pA[0];
   rC1_3 += rA1 * rB0;
   rA1 = pA[1];
   rC2_3 += rA2 * rB0;
   rA2 = pA[2];
   rC3_3 += rA3 * rB0;
   rA3 = pA[3];
   rC4_3 += rA4 * rB0;
   rA4 = pA[4];
```

```
         rC5_3 += rA5 * rB0;
         rA5 = pA[5];
         rC6_3 += rA6 * rB0;
         rA6 = pA[6];
         rC7_3 += rA7 * rB0;
         rA7 = pA[7];
         rC8_3 += rA8 * rB0;
         rA8 = pA[8];
         rC9_3 += rA9 * rB0;
         rA9 = pA[9];
         rC10_3 += rA10 * rB0;
         rA10 = pA[10];
         rC11_3 += rA11 * rB0;
         rA11 = pA[11];
         pA += 12;
      LOOP_END
KDONE:

       @iif ibet ! 0
               rC0_0m = pC[0];
         @iif ibet = 1
               rC0_0 = rC0_0 + rC0_0m;
         @endiif
         @iif ibet = -1
               rC0_0 = rC0_0 - rC0_0m;
         @endiif
       @endiif
               pC[0] = rC0_0;
       @iif ibet ! 0
               rC1_0m = pC[1];
         @iif ibet = 1
               rC1_0 = rC1_0 + rC1_0m;
         @endiif
         @iif ibet = -1
               rC1_0 = rC1_0 - rC1_0m;
         @endiif
       @endiif
               pC[1] = rC1_0;
       @iif ibet ! 0
               rC2_0m = pC[2];
         @iif ibet = 1
               rC2_0 = rC2_0 + rC2_0m;
         @endiif
         @iif ibet = -1
               rC2_0 = rC2_0 - rC2_0m;
         @endiif
       @endiif
               pC[2] = rC2_0;
       @iif ibet ! 0
               rC3_0m = pC[3];
         @iif ibet = 1
               rC3_0 = rC3_0 + rC3_0m;
         @endiif
         @iif ibet = -1
               rC3_0 = rC3_0 - rC3_0m;
         @endiif
       @endiif
               pC[3] = rC3_0;
       @iif ibet ! 0
               rC4_0m = pC[4];
         @iif ibet = 1
               rC4_0 = rC4_0 + rC4_0m;
         @endiif
         @iif ibet = -1
               rC4_0 = rC4_0 - rC4_0m;
         @endiif
       @endiif
               pC[4] = rC4_0;
       @iif ibet ! 0
               rC5_0m = pC[5];
         @iif ibet = 1
               rC5_0 = rC5_0 + rC5_0m;
         @endiif
         @iif ibet = -1
               rC5_0 = rC5_0 - rC5_0m;
         @endiif
       @endiif
               pC[5] = rC5_0;
       @iif ibet ! 0
               rC6_0m = pC[6];
         @iif ibet = 1
               rC6_0 = rC6_0 + rC6_0m;
         @endiif
         @iif ibet = -1
               rC6_0 = rC6_0 - rC6_0m;
         @endiif
       @endiif
               pC[6] = rC6_0;
       @iif ibet ! 0
               rC7_0m = pC[7];
         @iif ibet = 1
               rC7_0 = rC7_0 + rC7_0m;
         @endiif
         @iif ibet = -1
               rC7_0 = rC7_0 - rC7_0m;
         @endiif
       @endiif
               pC[7] = rC7_0;
       @iif ibet ! 0
               rC8_0m = pC[8];
         @iif ibet = 1
               rC8_0 = rC8_0 + rC8_0m;
         @endiif
         @iif ibet = -1
               rC8_0 = rC8_0 - rC8_0m;
         @endiif
       @endiif
               pC[8] = rC8_0;
       @iif ibet ! 0
               rC9_0m = pC[9];
         @iif ibet = 1
               rC9_0 = rC9_0 + rC9_0m;
         @endiif
         @iif ibet = -1
               rC9_0 = rC9_0 - rC9_0m;
         @endiif
       @endiif
               pC[9] = rC9_0;
       @iif ibet ! 0
               rC10_0m = pC[10];
         @iif ibet = 1
               rC10_0 = rC10_0 + rC10_0m;
         @endiif
         @iif ibet = -1
               rC10_0 = rC10_0 - rC10_0m;
         @endiif
       @endiif
               pC[10] = rC10_0;
       @iif ibet ! 0
               rC11_0m = pC[11];
         @iif ibet = 1
               rC11_0 = rC11_0 + rC11_0m;
         @endiif
         @iif ibet = -1
               rC11_0 = rC11_0 - rC11_0m;
         @endiif
```

```
@endiif
        pC[11] = rC11_0;
@iif ibet ! 0
        rC0_1m = pC[12];
  @iif ibet = 1
        rC0_1 = rC0_1 + rC0_1m;
  @endiif
  @iif ibet = -1
        rC0_1 = rC0_1 - rC0_1m;
  @endiif
@endiif
        pC[12] = rC0_1;
@iif ibet ! 0
        rC1_1m = pC[13];
  @iif ibet = 1
        rC1_1 = rC1_1 + rC1_1m;
  @endiif
  @iif ibet = -1
        rC1_1 = rC1_1 - rC1_1m;
  @endiif
@endiif
        pC[13] = rC1_1;
@iif ibet ! 0
        rC2_1m = pC[14];
  @iif ibet = 1
        rC2_1 = rC2_1 + rC2_1m;
  @endiif
  @iif ibet = -1
        rC2_1 = rC2_1 - rC2_1m;
  @endiif
@endiif
        pC[14] = rC2_1;
@iif ibet ! 0
        rC3_1m = pC[15];
  @iif ibet = 1
        rC3_1 = rC3_1 + rC3_1m;
  @endiif
  @iif ibet = -1
        rC3_1 = rC3_1 - rC3_1m;
  @endiif
@endiif
        pC[15] = rC3_1;
@iif ibet ! 0
        rC4_1m = pC[16];
  @iif ibet = 1
        rC4_1 = rC4_1 + rC4_1m;
  @endiif
  @iif ibet = -1
        rC4_1 = rC4_1 - rC4_1m;
  @endiif
@endiif
        pC[16] = rC4_1;
@iif ibet ! 0
        rC5_1m = pC[17];
  @iif ibet = 1
        rC5_1 = rC5_1 + rC5_1m;
  @endiif
  @iif ibet = -1
        rC5_1 = rC5_1 - rC5_1m;
  @endiif
@endiif
        pC[17] = rC5_1;
@iif ibet ! 0
        rC6_1m = pC[18];
  @iif ibet = 1
        rC6_1 = rC6_1 + rC6_1m;
  @endiif
  @iif ibet = -1
        rC6_1 = rC6_1 - rC6_1m;
  @endiif
@endiif
        pC[18] = rC6_1;
@iif ibet ! 0
        rC7_1m = pC[19];
  @iif ibet = 1
        rC7_1 = rC7_1 + rC7_1m;
  @endiif
  @iif ibet = -1
        rC7_1 = rC7_1 - rC7_1m;
  @endiif
@endiif
        pC[19] = rC7_1;
@iif ibet ! 0
        rC8_1m = pC[20];
  @iif ibet = 1
        rC8_1 = rC8_1 + rC8_1m;
  @endiif
  @iif ibet = -1
        rC8_1 = rC8_1 - rC8_1m;
  @endiif
@endiif
        pC[20] = rC8_1;
@iif ibet ! 0
        rC9_1m = pC[21];
  @iif ibet = 1
        rC9_1 = rC9_1 + rC9_1m;
  @endiif
  @iif ibet = -1
        rC9_1 = rC9_1 - rC9_1m;
  @endiif
@endiif
        pC[21] = rC9_1;
@iif ibet ! 0
        rC10_1m = pC[22];
  @iif ibet = 1
        rC10_1 = rC10_1 + rC10_1m;
  @endiif
  @iif ibet = -1
        rC10_1 = rC10_1 - rC10_1m;
  @endiif
@endiif
        pC[22] = rC10_1;
@iif ibet ! 0
        rC11_1m = pC[23];
  @iif ibet = 1
        rC11_1 = rC11_1 + rC11_1m;
  @endiif
  @iif ibet = -1
        rC11_1 = rC11_1 - rC11_1m;
  @endiif
@endiif
        pC[23] = rC11_1;
@iif ibet ! 0
        rC0_2m = pC[24];
  @iif ibet = 1
        rC0_2 = rC0_2 + rC0_2m;
  @endiif
  @iif ibet = -1
        rC0_2 = rC0_2 - rC0_2m;
  @endiif
@endiif
        pC[24] = rC0_2;
@iif ibet ! 0
        rC1_2m = pC[25];
  @iif ibet = 1
        rC1_2 = rC1_2 + rC1_2m;
```

```
  @endiif
  @iif ibet = -1
        rC1_2 = rC1_2 - rC1_2m;
  @endiif
@endiif
        pC[25] = rC1_2;
@iif ibet ! 0
        rC2_2m = pC[26];
  @iif ibet = 1
        rC2_2 = rC2_2 + rC2_2m;
  @endiif
  @iif ibet = -1
        rC2_2 = rC2_2 - rC2_2m;
  @endiif
@endiif
        pC[26] = rC2_2;
@iif ibet ! 0
        rC3_2m = pC[27];
  @iif ibet = 1
        rC3_2 = rC3_2 + rC3_2m;
  @endiif
  @iif ibet = -1
        rC3_2 = rC3_2 - rC3_2m;
  @endiif
@endiif
        pC[27] = rC3_2;
@iif ibet ! 0
        rC4_2m = pC[28];
  @iif ibet = 1
        rC4_2 = rC4_2 + rC4_2m;
  @endiif
  @iif ibet = -1
        rC4_2 = rC4_2 - rC4_2m;
  @endiif
@endiif
        pC[28] = rC4_2;
@iif ibet ! 0
        rC5_2m = pC[29];
  @iif ibet = 1
        rC5_2 = rC5_2 + rC5_2m;
  @endiif
  @iif ibet = -1
        rC5_2 = rC5_2 - rC5_2m;
  @endiif
@endiif
        pC[29] = rC5_2;
@iif ibet ! 0
        rC6_2m = pC[30];
  @iif ibet = 1
        rC6_2 = rC6_2 + rC6_2m;
  @endiif
  @iif ibet = -1
        rC6_2 = rC6_2 - rC6_2m;
  @endiif
@endiif
        pC[30] = rC6_2;
@iif ibet ! 0
        rC7_2m = pC[31];
  @iif ibet = 1
        rC7_2 = rC7_2 + rC7_2m;
  @endiif
  @iif ibet = -1
        rC7_2 = rC7_2 - rC7_2m;
  @endiif
@endiif
        pC[31] = rC7_2;
@iif ibet ! 0
        rC8_2m = pC[32];
  @iif ibet = 1
        rC8_2 = rC8_2 + rC8_2m;
  @endiif
  @iif ibet = -1
        rC8_2 = rC8_2 - rC8_2m;
  @endiif
@endiif
        pC[32] = rC8_2;
@iif ibet ! 0
        rC9_2m = pC[33];
  @iif ibet = 1
        rC9_2 = rC9_2 + rC9_2m;
  @endiif
  @iif ibet = -1
        rC9_2 = rC9_2 - rC9_2m;
  @endiif
@endiif
        pC[33] = rC9_2;
@iif ibet ! 0
        rC10_2m = pC[34];
  @iif ibet = 1
        rC10_2 = rC10_2 + rC10_2m;
  @endiif
  @iif ibet = -1
        rC10_2 = rC10_2 - rC10_2m;
  @endiif
@endiif
        pC[34] = rC10_2;
@iif ibet ! 0
        rC11_2m = pC[35];
  @iif ibet = 1
        rC11_2 = rC11_2 + rC11_2m;
  @endiif
  @iif ibet = -1
        rC11_2 = rC11_2 - rC11_2m;
  @endiif
@endiif
        pC[35] = rC11_2;
@iif ibet ! 0
        rC0_3m = pC[36];
  @iif ibet = 1
        rC0_3 = rC0_3 + rC0_3m;
  @endiif
  @iif ibet = -1
        rC0_3 = rC0_3 - rC0_3m;
  @endiif
@endiif
        pC[36] = rC0_3;
@iif ibet ! 0
        rC1_3m = pC[37];
  @iif ibet = 1
        rC1_3 = rC1_3 + rC1_3m;
  @endiif
  @iif ibet = -1
        rC1_3 = rC1_3 - rC1_3m;
  @endiif
@endiif
        pC[37] = rC1_3;
@iif ibet ! 0
        rC2_3m = pC[38];
  @iif ibet = 1
        rC2_3 = rC2_3 + rC2_3m;
  @endiif
  @iif ibet = -1
        rC2_3 = rC2_3 - rC2_3m;
  @endiif
@endiif
        pC[38] = rC2_3;
```

```
      @iif ibet ! 0
              rC3_3m = pC[39];
        @iif ibet = 1
              rC3_3 = rC3_3 + rC3_3m;
        @endiif
        @iif ibet = -1
              rC3_3 = rC3_3 - rC3_3m;
        @endiif
      @endiif
              pC[39] = rC3_3;
      @iif ibet ! 0
              rC4_3m = pC[40];
        @iif ibet = 1
              rC4_3 = rC4_3 + rC4_3m;
        @endiif
        @iif ibet = -1
              rC4_3 = rC4_3 - rC4_3m;
        @endiif
      @endiif
              pC[40] = rC4_3;
      @iif ibet ! 0
              rC5_3m = pC[41];
        @iif ibet = 1
              rC5_3 = rC5_3 + rC5_3m;
        @endiif
        @iif ibet = -1
              rC5_3 = rC5_3 - rC5_3m;
        @endiif
      @endiif
              pC[41] = rC5_3;
      @iif ibet ! 0
              rC6_3m = pC[42];
        @iif ibet = 1
              rC6_3 = rC6_3 + rC6_3m;
        @endiif
        @iif ibet = -1
              rC6_3 = rC6_3 - rC6_3m;
        @endiif
      @endiif
              pC[42] = rC6_3;
      @iif ibet ! 0
              rC7_3m = pC[43];
        @iif ibet = 1
              rC7_3 = rC7_3 + rC7_3m;
        @endiif
        @iif ibet = -1
              rC7_3 = rC7_3 - rC7_3m;
        @endiif
      @endiif
              pC[43] = rC7_3;
      @iif ibet ! 0
              rC8_3m = pC[44];
        @iif ibet = 1
              rC8_3 = rC8_3 + rC8_3m;
        @endiif
        @iif ibet = -1
              rC8_3 = rC8_3 - rC8_3m;
        @endiif
      @endiif
              pC[44] = rC8_3;
      @iif ibet ! 0
              rC9_3m = pC[45];
        @iif ibet = 1
              rC9_3 = rC9_3 + rC9_3m;
        @endiif
        @iif ibet = -1
              rC9_3 = rC9_3 - rC9_3m;
        @endiif
      @endiif
              pC[45] = rC9_3;
      @iif ibet ! 0
              rC10_3m = pC[46];
        @iif ibet = 1
              rC10_3 = rC10_3 + rC10_3m;
        @endiif
        @iif ibet = -1
              rC10_3 = rC10_3 - rC10_3m;
        @endiif
      @endiif
              pC[46] = rC10_3;
      @iif ibet ! 0
              rC11_3m = pC[47];
        @iif ibet = 1
              rC11_3 = rC11_3 + rC11_3m;
        @endiif
        @iif ibet = -1
              rC11_3 = rC11_3 - rC11_3m;
        @endiif
      @endiif
              pC[47] = rC11_3;
        pC += 48;
        pA = pA0;
        j = j - 1;
        IF (j > 0) GOTO NLOOP;
     pB = pB0;
     pA0 += incAm;
     pA = pA0;
     i = i - 1;
     IF (i > 0) GOTO MLOOP;
ROUT_END
```

A.2 Double Precision KVEC With $u_m = 12$ $u_n = 1$ $u_k = 4$

```
@ifdef ! ATL_MM_KB
   @define MMKB @0@
@endifdef
@ifdef  ATL_MM_KB
   @define MMKB @1@
@endifdef
@iif MMKB = 0
   @ifdef KB
      @iif KB > 0
         @define ATL_KBCONST @1@
         @define ATL_MM_KB @K@
      @endiif
      @iif KB { 0
         @define ATL_KBCONST @0@
         @define ATL_MM_KB @K@
      @endiif
   @endifdef
   @ifdef ! KB
     @define ATL_KBCONST @0@
     @define ATL_MM_KB @K@
   @endifdef
@endiif
@iif MMKB = 1
  @iif ATL_MM_KB > 0
      @define ATL_KBCONST @1@
  @endiif
```

```
  @iif ATL_MM_KB { 0
     @undef ATL_MM_KB
     @define ATL_MM_KB @K@
     @define ATL_KBCONST @0@
  @endiif
@endiif
@ifdef ! BETA1
   @ifdef ! BETA0
       @define ibet @-1@
   @endifdef
@endifdef
@ifdef BETA1
    @define ibet @1@
@endifdef
@ifdef BETA0
   @define ibet @0@
@endifdef


ROUTINE ATL_USERMM;
   PARAMS :: nmus, nnus, K, pA, pB, pC, pAn, pBn, pCn;
   INT :: nmus, nnus, K;
   DOUBLE_PTR :: pA, pB, pC, pAn, pBn, pCn;
//
// Performs a GEMM with M,N,K unrolling (& jam)
//    of (12,1,4).
// Vectorization of VLEN=4 along K dim, vec
//    unroll=(12,1,1).
// You may set compile-time constant K dim
//    by defining ATL_MM_KB.
//
ROUT_LOCALS
   DOUBLE :: rC0_0, rC1_0, rC2_0, rC3_0, rC4_0, rC5_0,
             rC6_0, rC7_0, rC8_0, rC9_0, rC10_0, rC11_0,
             rA0, rB0;
   DOUBLE_PTR :: pA0, pB0;
   INT :: i, j, k;
   INT :: incAm, incBn;
@iif ibet ! 0
  @declare "   DOUBLE :: " y n ";"
     @iexp j 0 0 +
     @iwhile j < 1
        @iexp i 0 0 +
        @iwhile i < 12
            rC@(i)_@(j)m
           @iexp i @(i) 1 +
        @endiwhile
        @iexp j @(j) 1 +
     @endiwhile
  @enddeclare
@endiif
ROUT_MARKUP
   ALIGNED(32) :: *;
ROUT_BEGIN
   pB0=pB;
   pA0=pA;
@iif ATL_KBCONST = 0
      incAm = K*12;
      incBn = K*1;
@endiif
@iif ATL_KBCONST ! 0
      incAm = @(ATL_MM_KB)*12;
      incBn = @(ATL_MM_KB)*1;
@endiif
   i = nmus;
   MLOOP:
         j =  nnus;
      NLOOP:
         rC0_0 = 0.0;
         rC1_0 = 0.0;
         rC2_0 = 0.0;
         rC3_0 = 0.0;
         rC4_0 = 0.0;
         rC5_0 = 0.0;
         rC6_0 = 0.0;
         rC7_0 = 0.0;
         rC8_0 = 0.0;
         rC9_0 = 0.0;
         rC10_0 = 0.0;
         rC11_0 = 0.0;
      LOOP k=0, @(ATL_MM_KB), 4
      LOOP_BODY
         rB0 = pB[0];
            rA0 = pA[0];
            rC0_0 += rA0 * rB0;
            rA0 = pA[4];
            rC1_0 += rA0 * rB0;
            rA0 = pA[8];
            rC2_0 += rA0 * rB0;
            rA0 = pA[12];
            rC3_0 += rA0 * rB0;
            rA0 = pA[16];
            rC4_0 += rA0 * rB0;
            rA0 = pA[20];
            rC5_0 += rA0 * rB0;
            rA0 = pA[24];
            rC6_0 += rA0 * rB0;
            rA0 = pA[28];
            rC7_0 += rA0 * rB0;
            rA0 = pA[32];
            rC8_0 += rA0 * rB0;
            rA0 = pA[36];
            rC9_0 += rA0 * rB0;
            rA0 = pA[40];
            rC10_0 += rA0 * rB0;
            rA0 = pA[44];
            rC11_0 += rA0 * rB0;
         rB0 = pB[1];
            rA0 = pA[1];
            rC0_0 += rA0 * rB0;
            rA0 = pA[5];
            rC1_0 += rA0 * rB0;
            rA0 = pA[9];
            rC2_0 += rA0 * rB0;
            rA0 = pA[13];
            rC3_0 += rA0 * rB0;
            rA0 = pA[17];
            rC4_0 += rA0 * rB0;
            rA0 = pA[21];
            rC5_0 += rA0 * rB0;
            rA0 = pA[25];
            rC6_0 += rA0 * rB0;
            rA0 = pA[29];
            rC7_0 += rA0 * rB0;
            rA0 = pA[33];
            rC8_0 += rA0 * rB0;
            rA0 = pA[37];
            rC9_0 += rA0 * rB0;
            rA0 = pA[41];
            rC10_0 += rA0 * rB0;
            rA0 = pA[45];
            rC11_0 += rA0 * rB0;
         rB0 = pB[2];
            rA0 = pA[2];
            rC0_0 += rA0 * rB0;
            rA0 = pA[6];
```

```
      rC1_0 += rA0 * rB0;
      rA0 = pA[10];
      rC2_0 += rA0 * rB0;
      rA0 = pA[14];
      rC3_0 += rA0 * rB0;
      rA0 = pA[18];
      rC4_0 += rA0 * rB0;
      rA0 = pA[22];
      rC5_0 += rA0 * rB0;
      rA0 = pA[26];
      rC6_0 += rA0 * rB0;
      rA0 = pA[30];
      rC7_0 += rA0 * rB0;
      rA0 = pA[34];
      rC8_0 += rA0 * rB0;
      rA0 = pA[38];
      rC9_0 += rA0 * rB0;
      rA0 = pA[42];
      rC10_0 += rA0 * rB0;
      rA0 = pA[46];
      rC11_0 += rA0 * rB0;
   rB0 = pB[3];
      rA0 = pA[3];
      rC0_0 += rA0 * rB0;
      rA0 = pA[7];
      rC1_0 += rA0 * rB0;
      rA0 = pA[11];
      rC2_0 += rA0 * rB0;
      rA0 = pA[15];
      rC3_0 += rA0 * rB0;
      rA0 = pA[19];
      rC4_0 += rA0 * rB0;
      rA0 = pA[23];
      rC5_0 += rA0 * rB0;
      rA0 = pA[27];
      rC6_0 += rA0 * rB0;
      rA0 = pA[31];
      rC7_0 += rA0 * rB0;
      rA0 = pA[35];
      rC8_0 += rA0 * rB0;
      rA0 = pA[39];
      rC9_0 += rA0 * rB0;
      rA0 = pA[43];
      rC10_0 += rA0 * rB0
      rA0 = pA[47];
      rC11_0 += rA0 * rB0;
      pA += 48;
      pB += 4;
LOOP_END
 @iif ibet ! 0
         rC0_0m = pC[0];
   @iif ibet = 1
         rC0_0 = rC0_0 + rC0_0m;
   @endiif
   @iif ibet = -1
         rC0_0 = rC0_0 - rC0_0m;
   @endiif
 @endiif
         pC[0] = rC0_0;
 @iif ibet ! 0
         rC1_0m = pC[1];
   @iif ibet = 1
         rC1_0 = rC1_0 + rC1_0m;
   @endiif
   @iif ibet = -1
         rC1_0 = rC1_0 - rC1_0m;
   @endiif
 @endiif
```

```
        pC[1] = rC1_0;
@iif ibet ! 0
        rC2_0m = pC[2];
  @iif ibet = 1
        rC2_0 = rC2_0 + rC2_0m;
  @endiif
  @iif ibet = -1
        rC2_0 = rC2_0 - rC2_0m;
  @endiif
@endiif
        pC[2] = rC2_0;
@iif ibet ! 0
        rC3_0m = pC[3];
  @iif ibet = 1
        rC3_0 = rC3_0 + rC3_0m;
  @endiif
  @iif ibet = -1
        rC3_0 = rC3_0 - rC3_0m;
  @endiif
@endiif
        pC[3] = rC3_0;
@iif ibet ! 0
        rC4_0m = pC[4];
  @iif ibet = 1
        rC4_0 = rC4_0 + rC4_0m;
  @endiif
  @iif ibet = -1
        rC4_0 = rC4_0 - rC4_0m;
  @endiif
@endiif
        pC[4] = rC4_0;
@iif ibet ! 0
        rC5_0m = pC[5];
  @iif ibet = 1
        rC5_0 = rC5_0 + rC5_0m;
  @endiif
  @iif ibet = -1
        rC5_0 = rC5_0 - rC5_0m;
  @endiif
@endiif
        pC[5] = rC5_0;
@iif ibet ! 0
        rC6_0m = pC[6];
  @iif ibet = 1
        rC6_0 = rC6_0 + rC6_0m;
  @endiif
  @iif ibet = -1
        rC6_0 = rC6_0 - rC6_0m;
  @endiif
@endiif
        pC[6] = rC6_0;
@iif ibet ! 0
        rC7_0m = pC[7];
  @iif ibet = 1
        rC7_0 = rC7_0 + rC7_0m;
  @endiif
  @iif ibet = -1
        rC7_0 = rC7_0 - rC7_0m;
  @endiif
@endiif
        pC[7] = rC7_0;
@iif ibet ! 0
        rC8_0m = pC[8];
  @iif ibet = 1
        rC8_0 = rC8_0 + rC8_0m;
  @endiif
  @iif ibet = -1
        rC8_0 = rC8_0 - rC8_0m;
```

```
  @endiif
@endiif
        pC[8] = rC8_0;
@iif ibet ! 0
        rC9_0m = pC[9];
  @iif ibet = 1
        rC9_0 = rC9_0 + rC9_0m;
  @endiif
  @iif ibet = -1
        rC9_0 = rC9_0 - rC9_0m;
  @endiif
@endiif
        pC[9] = rC9_0;
@iif ibet ! 0
        rC10_0m = pC[10];
  @iif ibet = 1
        rC10_0 = rC10_0 + rC10_0m;
  @endiif
  @iif ibet = -1
        rC10_0 = rC10_0 - rC10_0m;
  @endiif
@endiif
        pC[10] = rC10_0;
@iif ibet ! 0
        rC11_0m = pC[11];
  @iif ibet = 1
        rC11_0 = rC11_0 + rC11_0m;
  @endiif
  @iif ibet = -1
        rC11_0 = rC11_0 - rC11_0m;
  @endiif
@endiif
        pC[11] = rC11_0;
  pC += 12;
  pA = pA0;
  j = j - 1;
  IF (j > 0) GOTO NLOOP;
pB = pB0;
pA0 += incAm;
pA = pA0;
i = i - 1;
IF (i > 0) GOTO MLOOP;
ROUT_END
```

APPENDIX B
ATLAS GEMM MICROKERNELS IN C

This appendix provides the C implementation of the *gemmμ* scalar kernels given as input to the industrial compilers in Chapter 6:

- Section B.1 shows a scalar double precision *gemmμ* with $u_m = 12$, $u_n = 4$ and $u_k = 1$ that can be vectorized along the M dimension.
- Section B.2 provides a scalar double precision *gemmμ* implementation with $u_m = 12$, $u_n = 1$ and $u_k = 4$ that can be vectorized along the K dimension.

Finally, Section B.4 provides compiler versions, flags and pragmas used in the experiments described in Chapter 6.

B.1 Double Precision MVEC With $u_m = 12$, $u_n = 4$, $u_k = 1$

```
#ifndef ATL_MM_KB
   #ifdef KB
      #if KB > 0
         #define ATL_KBCONST 1
         #define ATL_MM_KB KB
      #else
         #define ATL_KBCONST 0
         #define ATL_MM_KB K
      #endif
   #else
      #define ATL_KBCONST 0
      #define ATL_MM_KB K
   #endif
#else
   #if ATL_MM_KB > 0
      #define ATL_KBCONST 1
   #else
      #undef ATL_MM_KB
      #define ATL_MM_KB K
      #define ATL_KBCONST 0
   #endif
#endif
#ifdef BETA1
   #define ATL_vbeta(p_, idx, d_) \
   { \
      rA0 = p_[idx]; \
      d_ = d_ + rA0; \
      p_[idx] = d_; \
   }
#elif defined(BETA0)
   #define ATL_vbeta(p_, idx, d_) p_[idx] = d_;
#else
   #define ATL_vbeta(p_, idx, d_) \
   { \
      rA0 = p_[idx]; \
      d_ = d_ - rA0; \
      p_[idx] = d_; \
   }
#endif
#ifndef ATL_CSZT
    #include <stddef.h>
    #define ATL_CSZT const size_t
#endif
void ATL_USERMM
(
   ATL_CSZT nmus,
   ATL_CSZT nnus,
   ATL_CSZT K,
#if defined(__clang__)
 const double *pA __attribute__((align_value(32))),
 const double *pB __attribute__((align_value(32))),
 double *restrict pC __attribute__((align_value(32))),
#else
 const double *pA,
 const double *pB,
 double *restrict pC,
#endif
   const double *pAn, /* next block of A */
   const double *pBn, /* next block of B */
   const double *pCn  /* next block of C */
)
//
// Performs a GEMM with M,N,K unrolling (& jam)
//    of (12,4,1).
// Vectorization of VLEN=1 along M dim,
//    vec unroll=(12,4,1).
// You may set compile-time constant K dim by
//    defining ATL_MM_KB.
//
{
   double rB0, rC0_0, rC1_0, rC2_0, rC3_0, rC4_0,rC5_0,
          rC6_0, rC7_0,  rC8_0, rC9_0,  rC10_0, rC11_0,
          rC0_1, rC1_1,  rC2_1, rC3_1,  rC4_1,  rC5_1,
          rC6_1, rC7_1,  rC8_1, rC9_1,  rC10_1, rC11_1,
          rC0_2, rC1_2,  rC2_2, rC3_2,  rC4_2,  rC5_2,
          rC6_2, rC7_2,  rC8_2, rC9_2,  rC10_2, rC11_2,
          rC0_3, rC1_3,  rC2_3, rC3_3,  rC4_3,  rC5_3,
          rC6_3, rC7_3,  rC8_3, rC9_3,  rC10_3, rC11_3,
          rA0, rA1, rA2, rA3, rA4, rA5, rA6, rA7,rA8,
          rA9, rA10, rA11;
   const double *pA0, *pB0;
   int i, j, k;
   int incAm, incBn;
#if defined(__ICC) || defined(__INTEL_COMPILER)
   __assume_aligned(pA, 32);
   __assume_aligned(pB, 32);
```

```
   __assume_aligned(pC, 32);
#endif
   pB0=pB;
   pA0=pA;
   #if ATL_KBCONST == 0
      incAm = K*12;
      incBn = K*4;
   #else
      incAm = (12*ATL_MM_KB);
      incBn = (4*ATL_MM_KB);
   #endif
#if defined(__ICC) || defined(__INTEL_COMPILER)
   #pragma vector always
#endif
   for (i=0; i < nmus; i++)
   {
      for (j=0; j < nnus; j++)
      {
         // Peel K=0 iteration to avoid zero
         // of rCxx and extra add
         rA0 = pA[0];
         rA1 = pA[1];
         rA2 = pA[2];
         rA3 = pA[3];
         rA4 = pA[4];
         rA5 = pA[5];
         rA6 = pA[6];
         rA7 = pA[7];
         rA8 = pA[8];
         rA9 = pA[9];
         rA10 = pA[10];
         rA11 = pA[11];
         pA += 12;
         rB0 = pB[0];
         rC0_0 = rA0 * rB0;
         rC1_0 = rA1 * rB0;
         rC2_0 = rA2 * rB0;
         rC3_0 = rA3 * rB0;
         rC4_0 = rA4 * rB0;
         rC5_0 = rA5 * rB0;
         rC6_0 = rA6 * rB0;
         rC7_0 = rA7 * rB0;
         rC8_0 = rA8 * rB0;
         rC9_0 = rA9 * rB0;
         rC10_0 = rA10 * rB0;
         rC11_0 = rA11 * rB0;
         rB0 = pB[1];
         rC0_1 = rA0 * rB0;
         rC1_1 = rA1 * rB0;
         rC2_1 = rA2 * rB0;
         rC3_1 = rA3 * rB0;
         rC4_1 = rA4 * rB0;
         rC5_1 = rA5 * rB0;
         rC6_1 = rA6 * rB0;
         rC7_1 = rA7 * rB0;
         rC8_1 = rA8 * rB0;
         rC9_1 = rA9 * rB0;
         rC10_1 = rA10 * rB0;
         rC11_1 = rA11 * rB0;
         rB0 = pB[2];
         rC0_2 = rA0 * rB0;
         rC1_2 = rA1 * rB0;
         rC2_2 = rA2 * rB0;
         rC3_2 = rA3 * rB0;
         rC4_2 = rA4 * rB0;
         rC5_2 = rA5 * rB0;
         rC6_2 = rA6 * rB0;
         rC7_2 = rA7 * rB0;
         rC8_2 = rA8 * rB0;
         rC9_2 = rA9 * rB0;
         rC10_2 = rA10 * rB0;
         rC11_2 = rA11 * rB0;
         rB0 = pB[3];
         pB += 4;
         rC0_3 = rA0 * rB0;
         rA0 = pA[0];
         rC1_3 = rA1 * rB0;
         rA1 = pA[1];
         rC2_3 = rA2 * rB0;
         rA2 = pA[2];
         rC3_3 = rA3 * rB0;
         rA3 = pA[3];
         rC4_3 = rA4 * rB0;
         rA4 = pA[4];
         rC5_3 = rA5 * rB0;
         rA5 = pA[5];
         rC6_3 = rA6 * rB0;
         rA6 = pA[6];
         rC7_3 = rA7 * rB0;
         rA7 = pA[7];
         rC8_3 = rA8 * rB0;
         rA8 = pA[8];
         rC9_3 = rA9 * rB0;
         rA9 = pA[9];
         rC10_3 = rA10 * rB0;
         rA10 = pA[10];
         rC11_3 = rA11 * rB0;
         rA11 = pA[11];
         pA += 12;
         for (k=1; k < ATL_MM_KB; k += 1)
         {
            rB0 = pB[0];
            rC0_0 += rA0 * rB0;
            rC1_0 += rA1 * rB0;
            rC2_0 += rA2 * rB0;
            rC3_0 += rA3 * rB0;
            rC4_0 += rA4 * rB0;
            rC5_0 += rA5 * rB0;
            rC6_0 += rA6 * rB0;
            rC7_0 += rA7 * rB0;
            rC8_0 += rA8 * rB0;
            rC9_0 += rA9 * rB0;
            rC10_0 += rA10 * rB0;
            rC11_0 += rA11 * rB0;
            rB0 = pB[1];
            rC0_1 += rA0 * rB0;
            rC1_1 += rA1 * rB0;
            rC2_1 += rA2 * rB0;
            rC3_1 += rA3 * rB0;
            rC4_1 += rA4 * rB0;
            rC5_1 += rA5 * rB0;
            rC6_1 += rA6 * rB0;
            rC7_1 += rA7 * rB0;
            rC8_1 += rA8 * rB0;
            rC9_1 += rA9 * rB0;
            rC10_1 += rA10 * rB0;
            rC11_1 += rA11 * rB0;
            rB0 = pB[2];
            rC0_2 += rA0 * rB0;
            rC1_2 += rA1 * rB0;
            rC2_2 += rA2 * rB0;
            rC3_2 += rA3 * rB0;
            rC4_2 += rA4 * rB0;
            rC5_2 += rA5 * rB0;
            rC6_2 += rA6 * rB0;
            rC7_2 += rA7 * rB0;
```

```
            rC8_2 += rA8 * rB0;
            rC9_2 += rA9 * rB0;
            rC10_2 += rA10 * rB0;
            rC11_2 += rA11 * rB0;
            rB0 = pB[3];
            pB += 4;
            rC0_3 += rA0 * rB0;
            rA0 = pA[0];
            rC1_3 += rA1 * rB0;
            rA1 = pA[1];
            rC2_3 += rA2 * rB0;
            rA2 = pA[2];
            rC3_3 += rA3 * rB0;
            rA3 = pA[3];
            rC4_3 += rA4 * rB0;
            rA4 = pA[4];
            rC5_3 += rA5 * rB0;
            rA5 = pA[5];
            rC6_3 += rA6 * rB0;
            rA6 = pA[6];
            rC7_3 += rA7 * rB0;
            rA7 = pA[7];
            rC8_3 += rA8 * rB0;
            rA8 = pA[8];
            rC9_3 += rA9 * rB0;
            rA9 = pA[9];
            rC10_3 += rA10 * rB0;
            rA10 = pA[10];
            rC11_3 += rA11 * rB0;
            rA11 = pA[11];
               pA += 12;
         }
         ATL_vbeta(pC, 0, rC0_0);
         ATL_vbeta(pC, 1, rC1_0);
         ATL_vbeta(pC, 2, rC2_0);
         ATL_vbeta(pC, 3, rC3_0);
         ATL_vbeta(pC, 4, rC4_0);
         ATL_vbeta(pC, 5, rC5_0);
         ATL_vbeta(pC, 6, rC6_0);
         ATL_vbeta(pC, 7, rC7_0);
         ATL_vbeta(pC, 8, rC8_0);
         ATL_vbeta(pC, 9, rC9_0);
         ATL_vbeta(pC, 10, rC10_0);
         ATL_vbeta(pC, 11, rC11_0);
         ATL_vbeta(pC, 12, rC0_1);
         ATL_vbeta(pC, 13, rC1_1);
         ATL_vbeta(pC, 14, rC2_1);
         ATL_vbeta(pC, 15, rC3_1);
         ATL_vbeta(pC, 16, rC4_1);
         ATL_vbeta(pC, 17, rC5_1);
         ATL_vbeta(pC, 18, rC6_1);
         ATL_vbeta(pC, 19, rC7_1);
         ATL_vbeta(pC, 20, rC8_1);
         ATL_vbeta(pC, 21, rC9_1);
         ATL_vbeta(pC, 22, rC10_1);
         ATL_vbeta(pC, 23, rC11_1);
         ATL_vbeta(pC, 24, rC0_2);
         ATL_vbeta(pC, 25, rC1_2);
         ATL_vbeta(pC, 26, rC2_2);
         ATL_vbeta(pC, 27, rC3_2);
         ATL_vbeta(pC, 28, rC4_2);
         ATL_vbeta(pC, 29, rC5_2);
         ATL_vbeta(pC, 30, rC6_2);
         ATL_vbeta(pC, 31, rC7_2);
         ATL_vbeta(pC, 32, rC8_2);
         ATL_vbeta(pC, 33, rC9_2);
         ATL_vbeta(pC, 34, rC10_2);
         ATL_vbeta(pC, 35, rC11_2);
         ATL_vbeta(pC, 36, rC0_3);
         ATL_vbeta(pC, 37, rC1_3);
         ATL_vbeta(pC, 38, rC2_3);
         ATL_vbeta(pC, 39, rC3_3);
         ATL_vbeta(pC, 40, rC4_3);
         ATL_vbeta(pC, 41, rC5_3);
         ATL_vbeta(pC, 42, rC6_3);
         ATL_vbeta(pC, 43, rC7_3);
         ATL_vbeta(pC, 44, rC8_3);
         ATL_vbeta(pC, 45, rC9_3);
         ATL_vbeta(pC, 46, rC10_3);
         ATL_vbeta(pC, 47, rC11_3);
         pC += 48;
         pA = pA0;
      }  /* end of loop over N */
      pB = pB0;
      pA0 += incAm;
      pA = pA0;
   }  /* end of loop over M */
}
```

B.2 Double Precision MVEC With $u_m = 12$, $u_n = 4$, $u_k = 1$ Without Hoisting Loads of pA

```
#ifndef ATL_MM_KB
   #ifdef KB
      #if KB > 0
         #define ATL_KBCONST 1
         #define ATL_MM_KB KB
      #else
         #define ATL_KBCONST 0
         #define ATL_MM_KB K
      #endif
   #else
      #define ATL_KBCONST 0
      #define ATL_MM_KB K
   #endif
#else
   #if ATL_MM_KB > 0
      #define ATL_KBCONST 1
   #else
      #undef ATL_MM_KB
      #define ATL_MM_KB K
      #define ATL_KBCONST 0
   #endif
#endif
#ifdef BETA1
   #define ATL_vbeta(p_, idx, d_) \
   { \
      rA0 = p_[idx]; \
      d_ = d_ + rA0; \
      p_[idx] = d_; \
   }
#elif defined(BETA0)
   #define ATL_vbeta(p_, idx, d_) p_[idx] = d_;
#else
   #define ATL_vbeta(p_, idx, d_) \
   { \
      rA0 = p_[idx]; \
      d_ = d_ - rA0; \
      p_[idx] = d_; \
   }
```

```
#endif

#ifndef ATL_CSZT
    #include <stddef.h>
    #define ATL_CSZT const size_t
#endif
void ATL_USERMM
(
   ATL_CSZT nmus,
   ATL_CSZT nnus,
   ATL_CSZT K,
#if defined(__clang__)
 const double *pA __attribute__((align_value(32))),
 const double *pB __attribute__((align_value(32))),
 double *restrict pC __attribute__((align_value(32))),
#else
 const double *pA,
 const double *pB,
 double *restrict pC,
#endif
   const double *pAn, /* next block of A */
   const double *pBn, /* next block of B */
   const double *pCn  /* next block of C */
)
//
// Performs a GEMM with M,N,K unrolling
//   (& jam) of (12,4,1).
// Vectorization of VLEN=1 along M dim,
//   vec unroll=(12,4,1).
// You may set compile-time constant K dim
//   by defining ATL_MM_KB.
//
{
   double rB0, rC0_0, rC1_0, rC2_0, rC3_0, rC4_0,
          rC5_0, rC6_0, rC7_0, rC8_0,rC9_0, rC10_0,
          rC11_0, rC0_1, rC1_1, rC2_1, rC3_1,rC4_1,
          rC5_1,rC6_1, rC7_1, rC8_1, rC9_1, rC10_1,
          rC11_1, rC0_2, rC1_2, rC2_2,rC3_2, rC4_2,
          rC5_2, rC6_2, rC7_2, rC8_2, rC9_2, rC10_2,
          rC11_2,rC0_3, rC1_3, rC2_3, rC3_3, rC4_3,
          rC5_3, rC6_3, rC7_3, rC8_3, rC9_3, rC10_3,
          rC11_3, rA0, rA1, rA2, rA3, rA4, rA5, rA6,
          rA7, rA8, rA9, rA10, rA11;
   const double *pA0, *pB0;
   int i, j, k;
   int incAm, incBn;

#if defined(__ICC) || defined(__INTEL_COMPILER)
   __assume_aligned(pA, 32);
   __assume_aligned(pB, 32);
   __assume_aligned(pC, 32);
#endif

   pB0=pB;
   pA0=pA;
   #if ATL_KBCONST == 0
      incAm = K*12;
      incBn = K*4;
   #else
      incAm = (12*ATL_MM_KB);
      incBn = (4*ATL_MM_KB);
   #endif

#if defined(__ICC) || defined(__INTEL_COMPILER)
   #pragma vector always
#endif
   for (i=0; i < nmus; i++)
   {
      for (j=0; j < nnus; j++)
      {
         // Peel K=0 iteration to avoid zero of
         // rCxx and extra add
         rA0 = pA[0];
         rA1 = pA[1];
         rA2 = pA[2];
         rA3 = pA[3];
         rA4 = pA[4];
         rA5 = pA[5];
         rA6 = pA[6];
         rA7 = pA[7];
         rA8 = pA[8];
         rA9 = pA[9];
         rA10 = pA[10];
         rA11 = pA[11];
         pA += 12;
         rB0 = pB[0];
         rC0_0 = rA0 * rB0;
         rC1_0 = rA1 * rB0;
         rC2_0 = rA2 * rB0;
         rC3_0 = rA3 * rB0;
         rC4_0 = rA4 * rB0;
         rC5_0 = rA5 * rB0;
         rC6_0 = rA6 * rB0;
         rC7_0 = rA7 * rB0;
         rC8_0 = rA8 * rB0;
         rC9_0 = rA9 * rB0;
         rC10_0 = rA10 * rB0;
         rC11_0 = rA11 * rB0;
         rB0 = pB[1];
         rC0_1 = rA0 * rB0;
         rC1_1 = rA1 * rB0;
         rC2_1 = rA2 * rB0;
         rC3_1 = rA3 * rB0;
         rC4_1 = rA4 * rB0;
         rC5_1 = rA5 * rB0;
         rC6_1 = rA6 * rB0;
         rC7_1 = rA7 * rB0;
         rC8_1 = rA8 * rB0;
         rC9_1 = rA9 * rB0;
         rC10_1 = rA10 * rB0;
         rC11_1 = rA11 * rB0;
         rB0 = pB[2];
         rC0_2 = rA0 * rB0;
         rC1_2 = rA1 * rB0;
         rC2_2 = rA2 * rB0;
         rC3_2 = rA3 * rB0;
         rC4_2 = rA4 * rB0;
         rC5_2 = rA5 * rB0;
         rC6_2 = rA6 * rB0;
         rC7_2 = rA7 * rB0;
         rC8_2 = rA8 * rB0;
         rC9_2 = rA9 * rB0;
         rC10_2 = rA10 * rB0;
         rC11_2 = rA11 * rB0;
         rB0 = pB[3];
         pB += 4;
         rC0_3 = rA0 * rB0;
         rC1_3 = rA1 * rB0;
         rC2_3 = rA2 * rB0;
         rC3_3 = rA3 * rB0;
         rC4_3 = rA4 * rB0;
         rC5_3 = rA5 * rB0;
         rC6_3 = rA6 * rB0;
         rC7_3 = rA7 * rB0;
         rC8_3 = rA8 * rB0;
         rC9_3 = rA9 * rB0;
```

```
rC10_3 = rA10 * rB0;
rC11_3 = rA11 * rB0;

for (k=1; k < ATL_MM_KB; k += 1)
{
   rA0 = pA[0];
   rA1 = pA[1];
   rA2 = pA[2];
   rA3 = pA[3];
   rA4 = pA[4];
   rA5 = pA[5];
   rA6 = pA[6];
   rA7 = pA[7];
   rA8 = pA[8];
   rA9 = pA[9];
   rA10 = pA[10];
   rA11 = pA[11];
      pA += 12;
      rB0 = pB[0];
   rC0_0 += rA0 * rB0;
   rC1_0 += rA1 * rB0;
   rC2_0 += rA2 * rB0;
   rC3_0 += rA3 * rB0;
   rC4_0 += rA4 * rB0;
   rC5_0 += rA5 * rB0;
   rC6_0 += rA6 * rB0;
   rC7_0 += rA7 * rB0;
   rC8_0 += rA8 * rB0;
   rC9_0 += rA9 * rB0;
   rC10_0 += rA10 * rB0;
   rC11_0 += rA11 * rB0;
      rB0 = pB[1];
   rC0_1 += rA0 * rB0;
   rC1_1 += rA1 * rB0;
   rC2_1 += rA2 * rB0;
   rC3_1 += rA3 * rB0;
   rC4_1 += rA4 * rB0;
   rC5_1 += rA5 * rB0;
   rC6_1 += rA6 * rB0;
   rC7_1 += rA7 * rB0;
   rC8_1 += rA8 * rB0;
   rC9_1 += rA9 * rB0;
   rC10_1 += rA10 * rB0;
   rC11_1 += rA11 * rB0;
      rB0 = pB[2];
   rC0_2 += rA0 * rB0;
   rC1_2 += rA1 * rB0;
   rC2_2 += rA2 * rB0;
   rC3_2 += rA3 * rB0;
   rC4_2 += rA4 * rB0;
   rC5_2 += rA5 * rB0;
   rC6_2 += rA6 * rB0;
   rC7_2 += rA7 * rB0;
   rC8_2 += rA8 * rB0;
   rC9_2 += rA9 * rB0;
   rC10_2 += rA10 * rB0;
   rC11_2 += rA11 * rB0;
      rB0 = pB[3];
      pB += 4;
   rC0_3 += rA0 * rB0;
   rC1_3 += rA1 * rB0;
   rC2_3 += rA2 * rB0;
   rC3_3 += rA3 * rB0;
   rC4_3 += rA4 * rB0;
```

```
         rC5_3 += rA5 * rB0;
         rC6_3 += rA6 * rB0;
         rC7_3 += rA7 * rB0;
         rC8_3 += rA8 * rB0;
         rC9_3 += rA9 * rB0;
         rC10_3 += rA10 * rB0;
         rC11_3 += rA11 * rB0;
      }
      ATL_vbeta(pC, 0, rC0_0);
      ATL_vbeta(pC, 1, rC1_0);
      ATL_vbeta(pC, 2, rC2_0);
      ATL_vbeta(pC, 3, rC3_0);
      ATL_vbeta(pC, 4, rC4_0);
      ATL_vbeta(pC, 5, rC5_0);
      ATL_vbeta(pC, 6, rC6_0);
      ATL_vbeta(pC, 7, rC7_0);
      ATL_vbeta(pC, 8, rC8_0);
      ATL_vbeta(pC, 9, rC9_0);
      ATL_vbeta(pC, 10, rC10_0);
      ATL_vbeta(pC, 11, rC11_0);
      ATL_vbeta(pC, 12, rC0_1);
      ATL_vbeta(pC, 13, rC1_1);
      ATL_vbeta(pC, 14, rC2_1);
      ATL_vbeta(pC, 15, rC3_1);
      ATL_vbeta(pC, 16, rC4_1);
      ATL_vbeta(pC, 17, rC5_1);
      ATL_vbeta(pC, 18, rC6_1);
      ATL_vbeta(pC, 19, rC7_1);
      ATL_vbeta(pC, 20, rC8_1);
      ATL_vbeta(pC, 21, rC9_1);
      ATL_vbeta(pC, 22, rC10_1);
      ATL_vbeta(pC, 23, rC11_1);
      ATL_vbeta(pC, 24, rC0_2);
      ATL_vbeta(pC, 25, rC1_2);
      ATL_vbeta(pC, 26, rC2_2);
      ATL_vbeta(pC, 27, rC3_2);
      ATL_vbeta(pC, 28, rC4_2);
      ATL_vbeta(pC, 29, rC5_2);
      ATL_vbeta(pC, 30, rC6_2);
      ATL_vbeta(pC, 31, rC7_2);
      ATL_vbeta(pC, 32, rC8_2);
      ATL_vbeta(pC, 33, rC9_2);
      ATL_vbeta(pC, 34, rC10_2);
      ATL_vbeta(pC, 35, rC11_2);
      ATL_vbeta(pC, 36, rC0_3);
      ATL_vbeta(pC, 37, rC1_3);
      ATL_vbeta(pC, 38, rC2_3);
      ATL_vbeta(pC, 39, rC3_3);
      ATL_vbeta(pC, 40, rC4_3);
      ATL_vbeta(pC, 41, rC5_3);
      ATL_vbeta(pC, 42, rC6_3);
      ATL_vbeta(pC, 43, rC7_3);
      ATL_vbeta(pC, 44, rC8_3);
      ATL_vbeta(pC, 45, rC9_3);
      ATL_vbeta(pC, 46, rC10_3);
      ATL_vbeta(pC, 47, rC11_3);
      pC += 48;
      pA = pA0;
   }  /* end of loop over N */
   pB = pB0;
   pA0 += incAm;
   pA = pA0;
}  /* end of loop over M */
}
```

B.3 Double Precision KVEC Kernel With $u_m = 12$, $u_n = 1$ and $u_k = 4$

```
#ifndef ATL_MM_KB
   #ifdef KB
      #if KB > 0
         #define ATL_KBCONST 1
         #define ATL_MM_KB KB
      #else
         #define ATL_KBCONST 0
         #define ATL_MM_KB K
      #endif
   #else
      #define ATL_KBCONST 0
      #define ATL_MM_KB K
   #endif
#else
   #if ATL_MM_KB > 0
      #define ATL_KBCONST 1
   #else
      #undef ATL_MM_KB
      #define ATL_MM_KB K
      #define ATL_KBCONST 0
   #endif
#endif
#ifdef BETA1
   #define ATL_vbeta(p_, idx, d_) \
   { \
      rA0 = p_[idx]; \
      d_ = d_ + rA0; \
      p_[idx] = d_; \
   }
#elif defined(BETA0)
   #define ATL_vbeta(p_, idx, d_) p_[idx] = d_;
#else
   #define ATL_vbeta(p_, idx, d_) \
   { \
      rA0 = p_[idx]; \
      d_ = d_ - rA0; \
      p_[idx] = d_; \
   }
#endif


#ifndef ATL_CSZT
    #include <stddef.h>
    #define ATL_CSZT const size_t
#endif
void ATL_USERMM
(
   ATL_CSZT nmus,
   ATL_CSZT nnus,
   ATL_CSZT K,
#if defined(__clang__)
 const double *pA __attribute__((align_value(32))),
 const double *pB __attribute__((align_value(32))),
 double *restrict pC __attribute__((align_value(32))),
#else
 const double *pA,
 const double *pB,
 double * restrict pC,
#endif
   const double *pAn, /* next block of A */
   const double *pBn, /* next block of B */
   const double *pCn  /* next block of C */
)
//
// Performs a GEMM with M,N,K unrolling (& jam)
//    of (12,1,4).
// Vectorization of VLEN=4 along K dim,
//    vec unroll=(12,1,1).
// You may set compile-time constant K dim
//    by defining ATL_MM_KB.
//
{
   double rC0_0, rC1_0, rC2_0, rC3_0, rC4_0, rC5_0,
         rC6_0, rC7_0, rC8_0, rC9_0, rC10_0, rC11_0,
         rA0, rB0;
   const double *pA0, *pB0;
   int i, j, k;
   int incAm, incBn;
#if defined(__ICC) || defined(__INTEL_COMPILER)
   __assume_aligned(pA, 32);
   __assume_aligned(pB, 32);
   __assume_aligned(pC, 32);
#endif
   pB0=pB;
   pA0=pA;
   #if ATL_KBCONST == 0
      incAm = K*12;
      incBn = K*1;
   #else
      incAm = (12*ATL_MM_KB);
      incBn = (1*ATL_MM_KB);
   #endif
#if defined(__ICC) || defined(__INTEL_COMPILER)
   #pragma vector always
#endif
   for (i=0; i < nmus; i++)
   {
      for (j=0; j < nnus; j++)
      {
         rC0_0 = 0.0;
         rC1_0 = 0.0;
         rC2_0 = 0.0;
         rC3_0 = 0.0;
         rC4_0 = 0.0;
         rC5_0 = 0.0;
         rC6_0 = 0.0;
         rC7_0 = 0.0;
         rC8_0 = 0.0;
         rC9_0 = 0.0;
         rC10_0 = 0.0;
         rC11_0 = 0.0;
         for (k=0; k < ATL_MM_KB; k += 4)
         {
         rB0 = pB[0];
            rA0 = pA[0];
            rC0_0 += rA0 * rB0;
            rA0 = pA[4];
            rC1_0 += rA0 * rB0;
            rA0 = pA[8];
            rC2_0 += rA0 * rB0;
            rA0 = pA[12];
            rC3_0 += rA0 * rB0;
            rA0 = pA[16];
            rC4_0 += rA0 * rB0;
            rA0 = pA[20];
            rC5_0 += rA0 * rB0;
            rA0 = pA[24];
            rC6_0 += rA0 * rB0;
            rA0 = pA[28];
            rC7_0 += rA0 * rB0;
```

```
      rA0 = pA[32];
      rC8_0 += rA0 * rB0;
      rA0 = pA[36];
      rC9_0 += rA0 * rB0;
      rA0 = pA[40];
      rC10_0 += rA0 * rB0;
      rA0 = pA[44];
      rC11_0 += rA0 * rB0;
   rB0 = pB[1];
      rA0 = pA[1];
      rC0_0 += rA0 * rB0;
      rA0 = pA[5];
      rC1_0 += rA0 * rB0;
      rA0 = pA[9];
      rC2_0 += rA0 * rB0;
      rA0 = pA[13];
      rC3_0 += rA0 * rB0;
      rA0 = pA[17];
      rC4_0 += rA0 * rB0;
      rA0 = pA[21];
      rC5_0 += rA0 * rB0;
      rA0 = pA[25];
      rC6_0 += rA0 * rB0;
      rA0 = pA[29];
      rC7_0 += rA0 * rB0;
      rA0 = pA[33];
      rC8_0 += rA0 * rB0;
      rA0 = pA[37];
      rC9_0 += rA0 * rB0;
      rA0 = pA[41];
      rC10_0 += rA0 * rB0
      rA0 = pA[45];
      rC11_0 += rA0 * rB0
   rB0 = pB[2];
      rA0 = pA[2];
      rC0_0 += rA0 * rB0;
      rA0 = pA[6];
      rC1_0 += rA0 * rB0;
      rA0 = pA[10];
      rC2_0 += rA0 * rB0;
      rA0 = pA[14];
      rC3_0 += rA0 * rB0;
      rA0 = pA[18];
      rC4_0 += rA0 * rB0;
      rA0 = pA[22];
      rC5_0 += rA0 * rB0;
      rA0 = pA[26];
      rC6_0 += rA0 * rB0;
      rA0 = pA[30];
      rC7_0 += rA0 * rB0;
      rA0 = pA[34];
      rC8_0 += rA0 * rB0;
      rA0 = pA[38];
      rC9_0 += rA0 * rB0;
      rA0 = pA[42];
      rC10_0 += rA0 * rB0;
      rA0 = pA[46];
      rC11_0 += rA0 * rB0;
   rB0 = pB[3];
      rA0 = pA[3];
      rC0_0 += rA0 * rB0;
      rA0 = pA[7];
      rC1_0 += rA0 * rB0;
      rA0 = pA[11];
      rC2_0 += rA0 * rB0;
      rA0 = pA[15];
      rC3_0 += rA0 * rB0;
      rA0 = pA[19];
      rC4_0 += rA0 * rB0;
      rA0 = pA[23];
      rC5_0 += rA0 * rB0;
      rA0 = pA[27];
      rC6_0 += rA0 * rB0;
      rA0 = pA[31];
      rC7_0 += rA0 * rB0;
      rA0 = pA[35];
      rC8_0 += rA0 * rB0;
      rA0 = pA[39];
      rC9_0 += rA0 * rB0;
      rA0 = pA[43];
      rC10_0 += rA0 * rB0;
      rA0 = pA[47];
      rC11_0 += rA0 * rB0;
   pA += 48;
   pB += 4;
   }
   ATL_vbeta(pC, 0, rC0_0);
   ATL_vbeta(pC, 1, rC1_0);
   ATL_vbeta(pC, 2, rC2_0);
   ATL_vbeta(pC, 3, rC3_0);
   ATL_vbeta(pC, 4, rC4_0);
   ATL_vbeta(pC, 5, rC5_0);
   ATL_vbeta(pC, 6, rC6_0);
   ATL_vbeta(pC, 7, rC7_0);
   ATL_vbeta(pC, 8, rC8_0);
   ATL_vbeta(pC, 9, rC9_0);
   ATL_vbeta(pC, 10, rC10_0);
   ATL_vbeta(pC, 11, rC11_0);
   pC += 12;
   pA = pA0;
  }  /* end of loop over N */
  pB = pB0;
  pA0 += incAm;
  pA = pA0;
 }  /* end of loop over M */
}
```

B.4 Flags and Pragmas Used to Autovectorize Kernels

We tried several combinations of compiler flags to find best possible vectorized code produced by the autovectorizations of different compilers (shown in Table B.1). In additional to the flags, we used *pragma* to guide compilers to vectorize loop and *attribute* to specify the alignment of the addresses (where possible). Table B.2 shows all the flags, pramga and attribute we used (combination) of our experiments.

Table B.1: Industry compilers and their versions we used in our experiment on Intel Haswell machine

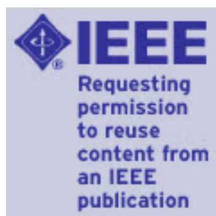
Compiler	version
ICC	17.0.2
GCC	5.4.0
CLANG+LLVM	4.0.0

Table B.2: Flags used to produce vectorize and scalar code

Compiler-Code_Type	Flags Used	pragma	special
FKO-Scalar	no-flag		
FKO-Vectorize	-vec		markup: ALIGNED(32)
ICC-Scalar	-O3/O2 -no-vec -march=core-avx2 -fomit-frame-pointer -m64		
ICC-Vectorize	-O3/O2 -vec -march=core-avx2 -fomit-frame-pointer -m64 -qopt-report=5	#pragma vector always	__assume_aligned (Ptr,32)
GCC-Scalar	-O3/O2 -fno-tree-vectorize -mavx2 -fomit-frame-pointer		
GCC-Vectorize	-O3/O2 -ftree-vectorize/-ftree-slp-vectorize -mfpmath=sse -mavx2 -mfma -ffast-math -fomit-frame-pointer -fopt-info-vec-all		
LLVM-Scalar	-O3/O2 -fno-vectorize -ffast-math -fomit-frame-pointer -mavx2 -mfma -m64		
LLVM-Vectorize	-O3/O2 -fslp-vectorize/-fslp-vectprize-aggressive -ffast-math -fomit-frame-pointer -mfpmath=sse -mavx2 -mfma -m64 -Rpass-analysis=loop-vectorize	#pragma clang loop vectorize(enable)	__attribute__((aligne_value(32)))

APPENDIX C

COPYRIGHT PERMISSIONS



Title: Vectorization past dependent branches through speculation
Conference Proceedings: Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on
Author: Majedul Haque Sujon
Publisher: IEEE
Date: Sept. 2013
Copyright © 2013, IEEE

LOGIN
If you're a copyright.com user, you can login to RightsLink using your copyright.com credentials. Already **a RightsLink user** or want to learn more?

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line � 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line � [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author�s approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: � [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Copyright © 2017 Copyright Clearance Center, Inc. All Rights Reserved. Privacy statement. Terms and Conditions.
Comments? We would like to hear from you. E-mail us at customercare@copyright.com

VITA

Md Majedul Haque Sujon was born on December 1981, in Tangail city, Bangladesh. He finished his undergraduate studies in Computer Science and Engineering at Bangladesh University of Engineering and Technology in June 2005. After working few years in industry, Majedul went back to school for his higher studies. He earned a master of science degree in computer science from University of Texas at San Antonio in August 2013. In August 2013 he came to Louisiana State University to pursue graduate studies in computer science. He is currently a candidate for the degree of Doctor of Philosophy in Computer Science and Engineering, which will be awarded in August 2017.