Automated Scratchpad Mapping and Allocation for Embedded Processors

by

Yang Gao

Bachelor of Science
Fuzhou University 2004

Master of Science
Shanghai Jiaotong University 2007

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2014

Accepted by:

Jason D. Bakos, Major Professor

Jianjun Hu, Committee Member

Manton M. Matthews, Committee Member

Yan Tong, Committee Member

Phil Moore, Committee Member

Lacy Ford, Vice Provost and Dean of Graduate Studies

Abstract

Embedded system-on-chip processors such as the Texas Instruments C66 DSP and the IBM Cell provide the programmer with a software controlled on-chip memory to supplement a traditional but simple two-level cache. By decomposing data sets and their corresponding workload into small subsets that fit within this on-chip memory, the processor can potentially achieve equivalent or better performance, power efficiency, and area efficiency than with its sophisticated cache. However, program controlled on chip memory requires a shift in the responsibility for management and allocation from the hardware to the programmer. Specifically, this requires the explicit mapping of program arrays to specific types of on chip memory structure and the addition of supporting code that allocates and manages the on chip memory. Previous work in tiling focuses on automated loop transformations but are hardware agnostic and do not incorporate a performance model of the underlying memory design. In this work we will explore the relationship between mapping and allocation of tiles for stencil loops and linear algebra kernels on the Texas Instruments Keystone II DSP platform.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Scratchpad memory (SPM) was originally designed as a way to avoid the non-deterministic performance of cache for hard real-time system designs [15]. SPM is implemented as high speed on-chip SRAM which guarantees an equivalent or better performance as cache-hit, but with less total energy consumption [3] [27].

Recent research shows that even in terms of performance, the SPM outperforms the cache. This is due to several reasons.

• The management of SPM is not based on data replacement. It is coupled with the core's behaviour and this avoids conflict misses. Those cache misses may dramatically undermine the performance of modern processors even with more than 8-way associative caches [24].

• To hide the read/write latencies, in a cache system, a prefetching mechanism is introduced to speculate the potential traffic according to past data access patterns [2]. However, this has no guarantee of sustainable utilization of the main memory bandwidth. Once the data are mistakenly prefetched, the system performance suffers from not only the high latency of the relaunched memory access but also the bandwidth wastage. The scratchpad with low access latency and high access bandwidth would minimize the loss from mis-prefetech.

• Once a cache miss occurs, the traditional CPU needs to stall the pipeline to wait for cache line to be filled. An non-blocking execution could alleviate this penalty by the out-of-order design, but the cost of this is system complexity and a corresponding degradation in terms of power-efficiency. Whereas, in most cases, the SPM-based

processors only need to maintain the simple in-order execution and use asynchronous transfers to hide memory latency.

The SPM has several disadvantages. While it gives the programmer flexibility with on-chip buffer allocation and management, the question about how to convert this flexibility to real performance is unclear. The other concern is that adding SPM control also increases the code size, complexity, and makes it more prone to errors.

Loop tiling with SPM is one of the best solutions to this problem. By tiling the outer loop into a series of inner loops, the data arrays are usually broken into small chunks with high locality. These chunks could then be easily mapped to the on-chip SPM and the increased data locality would also make it possible to access the main memory with continuous transfers facilitating the use of Direct Memory Access (DMA). In addition, the access pattern like this bulky transfer are more favourable due to the characteristic of DRAM memories.

Loop tiling method has been intensively researched in past few decades. The existing tools mainly target the implementation of an automatic compiler framework, which the program could be translated to a tiled version with very little or even without the programmer's assistance. Aside from Intel x86/x64 CPU platforms, some work have also been adapted to specific hardware platforms in order to benefit from the architectural features like shared memory in CUDA enabled GPUs. Those methods will be covered in the following chapters. However, the motivation of this research arises from the observation that previous works addressed the problem without a systematically considering the relationship between the feature of loop tiling kernels and the hardware specific information. As shown in Figure 1.1, when processing our testing kernels, the average performance of configurations with SPM is generally better than cache-only. Our optimized configuration could easily boost the system performance by another 50% with merely mapping to different on-chip buffer locations. With the models presented, we get similar speedup on most loop-intensive

kernels such as stencil loops or linear algebra kernels without any hand-tune efforts.

In this dissertation, we address this problem by combining a source-to-source translator with a performance model which will solve this problem in an automatic way. We choose the Texas Instruments Keystone II DSP as our experimental platform due to its highly flexible and reconfigurable memory hierarchy. Our proposed framework will compute the optimized working array mapping according to the data array access pattern, the hardware-specific information from processor specification and our statistical measurements. Our contributions are listed below:

•The mathematical abstraction to describe the DSP's memory system.

•A model on array access patterns in loop intensive kernels to ease the code translation targeting loop tiling implementations.

•The combination of these models to derive the optimized utilization and management of the DSP's on-chip SPM and EDMA resources.

•Verification of this model using stencil loop and linear algebra kernels.

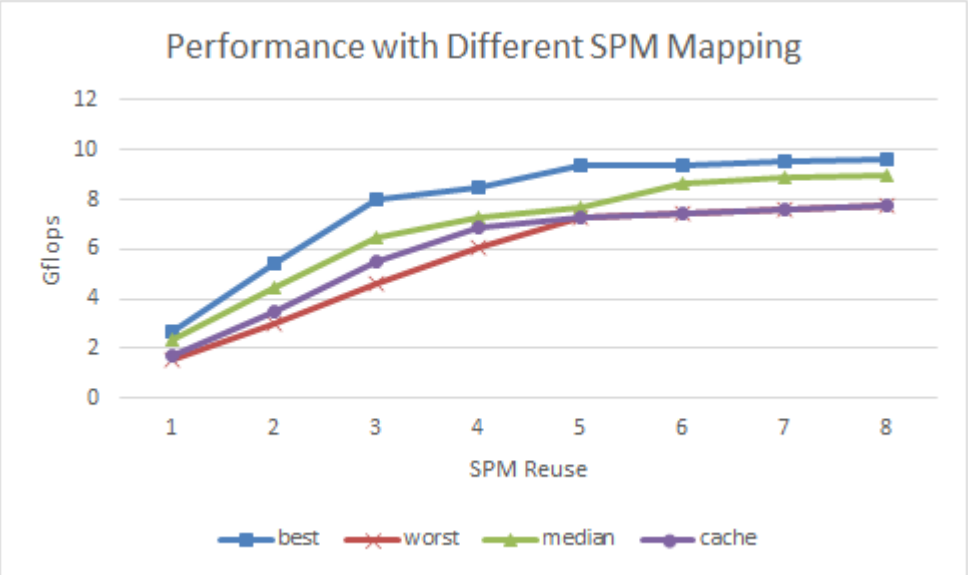

Figure 1.1: Performance with Different SPM Mapping. SPM Reuse indicates the frequency of SPM Buffer being accessed, a relative index to compute intensity.

The remainder of this document is organized as follows: in Chapter 2, we introduce

the Keystone II DSP, stencil loops and DMA/SPM buffering techniques. In Chapter 3, various methods to tile the stencils loop and linear algebra kernels are presented. The kernel motivated this work will also be covered in this chapter. Chapter 4 describes the statement of the problem. The methodology and the performance model is elaborated in Chapter 5. Chapter 6 provides our experimental results comparing with the ground truth results. Chapter 7 provide a summary and conclusion.

Chapter 2

Background

In this chapter, we present an overview of the DSP architecture and the related topics such as EDMA, SPM buffering and prefetch buffer. Besides these concepts, close attention has been paid to the DSP memory system performance, since it's the basis of our performance model. A selection of kernels we used to developed our methodology are cover at the last part.

2.1 Texas Instruments Keystone II DSP

As shown in Figure 2.1, the latest KeyStone II DSP architecture integrates four Cortex-A15 processors and eight C66x DSP cores. They are in two different logical structures called **ARM Corepac** and **DSP Corepac**. In this research, we are only interested in the performance of the DSP cores, since they are designed more for performance while the ARM are intended for control and inter-chip collaboration tasks. All the data traffic to/from the main DDR controller (DDRA) will be routed through the Multicore Shared Memory Controller (MSMC). Besides DDR, at most 6MB on-chip share memory is also hosted by the MSMC and available to all 8 DSP cores.

The C66x DSP Corepac design has several features that could make it highly power-efficient when used with carefully optimized and paralleled code:

- It lacks power-hungry features such as out-of-order and speculative execution and instead exploits instruction level parallelism using an eight-way very long

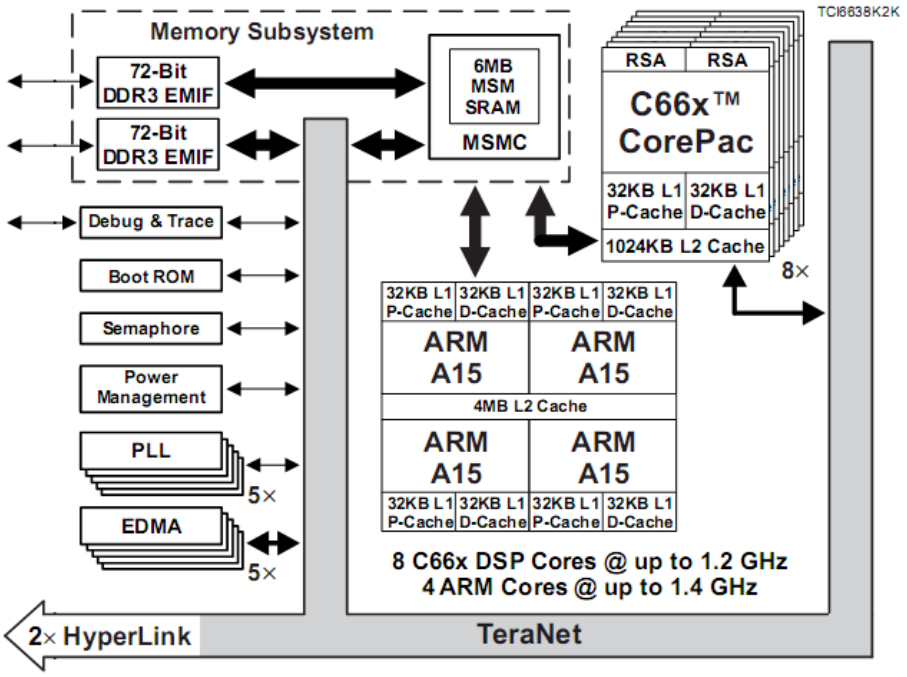

Figure 2.1: KeyStone II DSP Functional Block Diagram [28]

instruction word (VLIW).

- Its eight on-chip cores are loosely-coupled, as they do not include coherent mid-level caches nor a shared last-level cache. Instead it relies on explicit inter-core communication to exploit core-level parallelism.

- Each core has two levels of cache, but the caches can be reconfigured by the software such that a portion or all of one or both level of caches can be used as a software controlled scratchpad memory.

Table 2.1 summarizes the maximum theoretical throughput of different memories when the C66x device is operating at 1 GHz. The DDR3 performance assumes that a 64-bit bus width is used and that the external memory is operating at 1600 MHz. Notice that since the L1 and L2 could be configured as either cache, SPM, or both, (the available SPM capacity is the difference between total capacity and the cache). The MSMC could be configured as either level 2 or level 3 on-chip memory.

Table 2.1: KeyStone II DSP Memory Hierarchy and Performance

	Total Capacity	Cache Capacity	Cache Mode	Memory Level	Bandwidth GB/s	DSP Stalls[4] Cycles
L1P	32KB	0∼32KB[1]	Direct Mapped	1	32	0
L1D	32KB	0∼32KB[2]	2-Way	1	32	0
L2	1024KB	0∼1024KB[3]	4-Way	2	16	3.5
MSMC	6MB(8-core)	0	n/a	2 or 3	16×8	7.4
DDR	n/a	n/a	n/a	3	12.8	30.7

1: available size(KB)={0,4,8,16,32}
2: available size(KB)={0,4,8,16,32}
3: available size(KB)={0,32,64,128,256,512,1024}
4: average cycles when burst read with upper level cache miss and prefetch hit [29].

2.2 Direct Memory Access (DMA)

Direct memory access (DMA) is a feature of modern computers that allows certain hardware subsystems within the computer to access system memory independently of the central processing unit (CPU). Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work. With DMA, the CPU initiates the transfer, does other operations while the transfer is in progress, and receives an interrupt from the DMA controller when the operation is done.

If the per-core on-chip memory or the share memory (MSMC) are programmatically configured to behave partially or fully as a scratchpad memory (SPM) space, integrated DMA controller allows data to be exchanged between on- and off-chip memories in parallel to operations being performed on the DSP. The Enhanced Direct Memory Access (EDMA) controller can also be programmed to perform complex 3-dimensional data access patterns on both the source and destination memories to allow highly flexible transactions.

With EDMA, data accessed in a predictable, regular access pattern can be concurrently prefetched using EDMA to exchange data between scratchpad and DRAM.

Data accessed irregularly in a data-dependent pattern can be cached in a separate region of scratchpad to take advantage of locality. The usage of DMA for SPM buffering is a distinctive feature of TI DSP. Because in other CPUs with DMA, they are generally designed to handle the interaction with system I/O.

2.3 SPM Buffering Techniques

With DMA, a scratchpad memory(SPM) buffering technique can be applied to the computing kernel to overlap the off-chip memory content transfer and on-chip computing. To be more specific, in single buffering, the EDMA loads flushes the SPM without the DSP simultaneously processing the data. In double buffering, while the current block of data is being processed, the next block is being transferred.

Figure 2.2 Shows an example when TI DSP computes a memory bound kernel with double buffer technique. The figure only shows the case when reading. Likewise, when a kernel writes a data structure, this allows a block of on-chip data to be rendered by the DSP while the previous block is being written to off-chip memory.

Single and double buffer can both be beneficial the data access efficiency by reducing the cache miss penalty and increasing the per-transfer packet size from a single cache line to the SPM buffer size. In most cases, double buffering is more efficient due to the balance of computation time and data transmission time, but require more complexity.

The single buffer scheme takes more cycles due to serialization of computation and communication. However, this performance loss may be covered by the multi-core system. For instance, when core 1 is processing its on-chip data structure, core 2 may take usage of the EDMA engine loading from DDR to its on-chip SPM. If the data request to DDR exceed its maximal bandwidth capacity, the EDMA transfer would need to be queued. In this case, since the two buffering techniques has the same amount of EDMA transactions, when using multiple cores, there are almost no

difference between the two in terms of performance with multiple core processing.

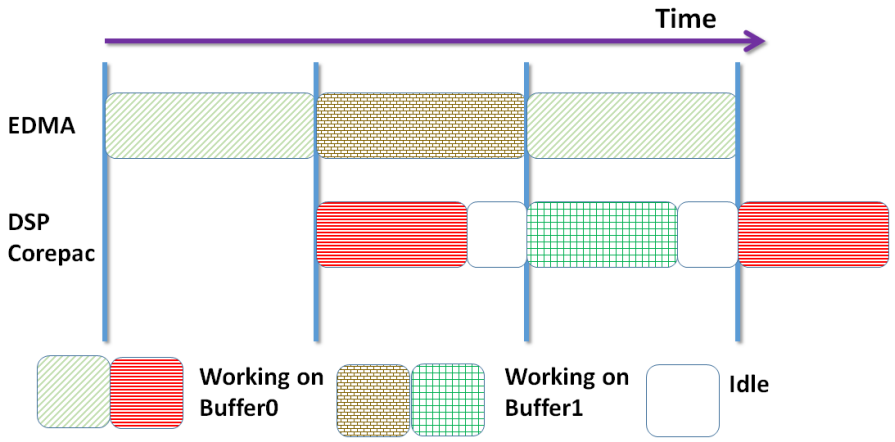

Figure 2.2: Double Buffer on TI DSP

2.4 Memory Throughput Analysis on DSP

The throughput of each type of SPM is bounded by the bandwidth of the DSP internal interconnections. As shown in Figure 2.3, the L2 features a 19.2 GB/s data path to DSP execution units, but only 6.4 GB/s to **External Memory Controller**(EMC) which is connected to DDR by the system bus. Since the DDR could provide a bandwidth up to 12.8 GB/s which is twice the speed of EMC to L2. So the bottleneck of the EDMA transactions between DDR and L2 is the EMC.

In the other hand, the bandwidth between MSMC and DDR is much higher according to Figure 2.3. And the MSMC is also connected to DSP execution units with a channel as fast as L2. However, this channel need to pass an internal memory controller in the DSP Corepac called **Extended Memory Controller**(XMC). This lead to extra latency when accessing MSMC.

So far we depicted the whole picture of the internal topology of the DSP. The bandwidth specification would help us make qualitative analysis to direct the SPM mapping in our following research. However, to make decision, we need more accurate data for each type of SPM when going through different path.

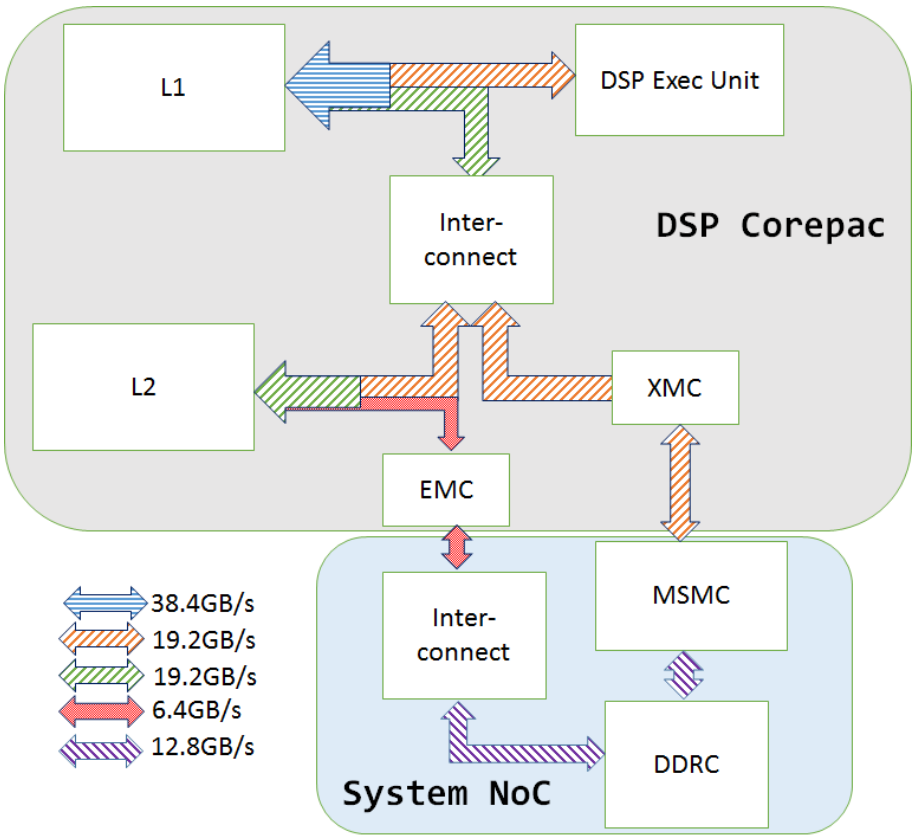

Figure 2.3: DSP on-chip Topology

2.5 Prefetch Buffer and DSP Stalls

The DSP contains a simple in-order pipeline. Whenever the cache misses occurs, it will stall and wait for data back. The length of cache miss stall depends on the access latency to the next level memory. For instance, suppose the access latency of L2 is 7 cycles, once the L1 cache missed on some address in L2 SPM or cached by L2, the DSP needs to stall for at least 7 cycles. If the L2 cache is also missed and data requirement need to go further to MSMC or DDR, more latencies are required.

We summarize the DSP stalls for in Table 2.2. In this table, we could find the L1 is free of access latency since it's running as the same frequency of the DSP. L2 is slower than L1 but better than MSMC and DDR because it's inside the Corepac and near to the L1 physically. Accesses to DDR requires the longest latency. In order to hide the latency and make better usage of the bandwidth, the DSP provided a

Table 2.2: DSP Stall Cycles on Cache Miss

Source	L1 Cache	Prefetch	DSP Stalls(in Cycles)
All	Hit	NA	0
Local L2 SRAM	Miss	NA	3.5
MSMC	Miss	Hit	7.4
MSMC	Miss	Miss	9.5
DDR	Miss	Hit	23.2
DDR	Miss	Miss	41.5

prefetch buffer inside the XMC to help the Corepac get data from outside.

The behaviour of this module is very simple. Once it detects the current address to be accessed, it reads the requested data as well as additional data that is expected to be required for a future miss. It contains 8 buffers which may track 8 memory access instances. This design significantly decreases the access latency to DDR and MSMC, especially for the compulsory and capacity cache misses. Since it contains more buffers than cache ways, it would also help to reduce the overhead of conflict cache misses.

2.6 Cache Performance Profiling Schemes

Analytical tools can be used to predict cache miss rate, such as cache miss equation[14][13] and polyhedral based methods[6]. Though, these models are able to solve miss rate in an analytic way, they are have many restrictions.

The profiling methods get the performance indices by running the kernel. On our TI DSP platform, we have two options to profile the code. One is through the cycle accurate simulation model. And the other one is by collecting results from the performance counter after running program on real hardware. Both of the methods are also summarized in Table 2.3.

Table 2.3: Ways to Measure the Cache and Prefetch Hit/Miss

	L1 miss rate	L2 miss rate	Prefetch h/m rate
Polyhedral model Cache Miss equations	Complexity. Not all kinds of cache misses are supported. Not 100% match hardware.		Not supported
Cycle accurate simulator	Support. Very slow. Discontinued in TI DSP Development IDE CCS 6.x Not 100% guarantee match hardware.		
On-chip performance counter	Not supported	Not supported	Support, Fast, Accurate, Need to run the Kernel

2.7 Stencil Loops

As shown in Figure 2.4, one type of our target kernels are stencil loops. Stencil loops are widely used in image processing, data mining, and physical simulations. they usually operate on multi-dimensional arrays, with each element computed as a function of neighbour elements.

As in Figure 2.5, generally, stencil loops have very high spacial and temporal locality. For this 8-point stencil, the values in cells marked **S** will be reused in the iterations in the same row. Row **T** will be reused in later rows. As a result, the key to improve performance resides in how to maximize reuse of these values while avoiding unnecessary data access from the main memory.

Another important characteristic of stencil loop is the "halo region" or "border effect". Since the values are decided by its neighbours, most stencil loop would start from the cells not residing in the frame edges, shown as the black cell in Figure 2.5. All greyed cells will only contribute as neighbours and will not get updated by the stencil loop. This does not play a vital role in a fully cached processor. However,

if the whole frame is split into tiles to fit in the SPM, the width of the halo region could be a decisive performance factor.

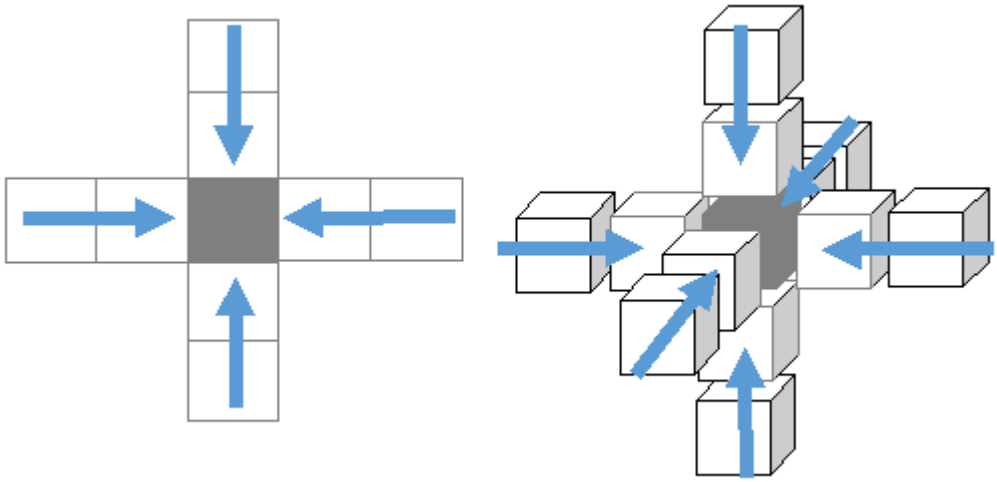

Figure 2.4: 8-point 2D stencil and 12-point 3D stencil

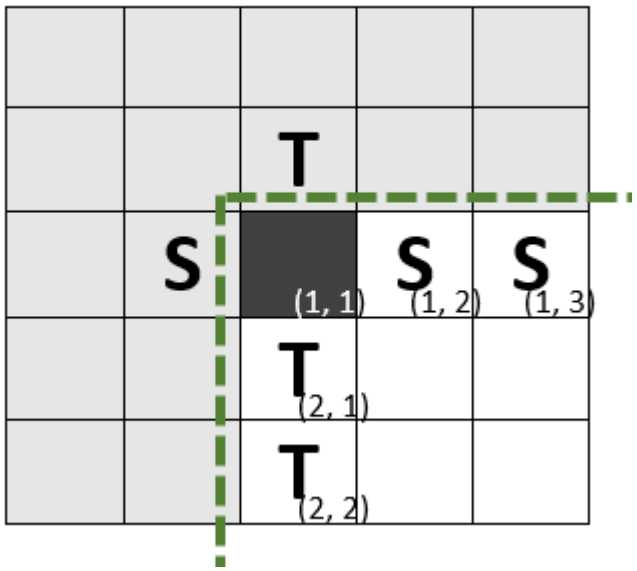

Figure 2.5: Data Locality and Halo Region in a 2D stencil

2.8 Sparse Matrix Vector Multiplication Kernel

Sparse Matrix Vector Multiplication (SpMV) is another of our target benchmarks. As a linear algebra kernel, it performs the computation $Y = A\alpha X + \beta Y$, where

A is a matrix stored in a sparse format, X and Y are vectors stored as dense 1D arrays, and α and β are scalars. Our SpMV kernel uses the popular Compressed Sparse Row (CSR) sparse matrix format, where matrix A is represented using three one-dimensional arrays, **val**, **col**, and **ptr**. The **val** array holds each of the matrix's non-zero values in ascending column and row order, while the **col** array holds each value's corresponding column index. The **ptr** array is indexed by row and holds the position within the **val** and **col** array where each matrix row begins. For example, an M x N matrix where M = 2 could be stored using arrays: $val = \{2, 4, 6, 8, 10, 12\}$, $col = \{2, 3, 4, 5, 3, 5\}$, and $ptr = \{0, 4, 6\}$. In this case, the matrix contains $ptr[M] = 6$ nonzero elements, the second row contains $ptr[2] - ptr[1] = 2$ elements, and the second element of row 1 is $val[ptr[1] + 1] = 12$ in column $col[ptr[1] + 1] = 5$. There are several reasons why sparse matrix-vector multiply with CSR format is a notoriously difficult kernel for which to achieve high functional unit utilization. First, the **col** array imposes gather-style indirect references to the input vector X, and the locality of the irregular accesses to X depends on the distribution of populated columns (defined in the col array). Second, the unpredictable number of entries per matrix row, as defined by the **ptr** array, requires dynamic control behaviour when computing the reduction operation when accumulating the inner product. Third, the entire operation is generally memory-bound for modern processors, requiring roughly 3/8 floating point operations per byte for single precision values and 32-bit indices, where n is average number of entries per matrix row.

Chapter 3

Previous Work

In this chapter, we summarize related work in scrachpad memory and loop tiling.

3.1 Scratchpad Method

Scratchpad memory (SPM) was initially introduced to improve energy efficiency [27] and timing responsiveness [15]. Often the objective of SPM research is related to partitioning the SPM and map data structure to it [10][32][31]. More recently, the IBM Cell processor increased interest in DMA-fed SPM. The Cell processor used its SPM explicitly to improve performance over cache. Intensive research has been done to explore its value on scientific computation.

Williams et al. [34] provided solutions to multiple kernels on Cell processor including stencil loops and linear algebra. They compared the tiled kernels performance to traditional cached counterparts. They made significant improvement in terms of both speed-up and power efficiency. More applications requiring platform dependent tuning are mapped to the Cell processors, such as dense linear algebra [17][20], sparse matrix-vector multiplication (SpMV) [33] , linear equations solving [19][18] and stencil computation [8][16]. Comparing to more popular architectures such as AMD Opteron, Intel Xeon, and Itanium, these methods achieve at least 2x speedup for memory-bound kernels and 7x to 150x for compute-bound ones.

TI DSP has a similar SPM characteristic to the Cell. The dense and sparse linear algebra kernels have been studied and proved effective with platform dependent SPM method [1][11][12]. Examples of effective usage of SPM in the literature are mostly

application-dependent. In these works, the loop tiling method has proven to be one of the most practical ways to utilize the SPM and DMA resources inside the chip. However, implementation a new kernel still generally requires the programmer to manually compose the kernel.

3.2 Automatic Loop Tiling Method

Loop tiling method seeks to reduce cache misses or increase SPM allocation efficiency. To explore the locality, the accesses in the iteration space need to be split into tiles to fit in high speed on-chip SRAM including either SPM or cache. Much research has been devoted to automate tiling, often using polyhedral frameworks [9][23][22]. Using a mathematical abstraction of the loop nest, the loop tilling method can be generalized to all processors.

Pluto [5] creatively introduced a new scheme to automate the loop tiling by optimizing the problem using a cost function. The most important contribution of this work is an end-to-end automatic parallelization and locality optimization for affine programs on general-purpose multi-core platforms. Its speedup ranging from 2x to 5x are achieved in general, and an order of 10x in best cases. Though it has been widely adopted and proved successfully under most platforms, due to its generality, when applied to non-traditional computing platforms like GPU, DSP, FPGA or other heterogeneous computing platforms, more platform specific characteristic need to be taken into consideration.

3.3 Automatic Loop Tiling with SPM

GPUs are popular target platform for automatic loop tilling frameworks. C-to-CUDA [4] is an automatic source-to-source polyhedral compiler for GPU targets based on Pluto. It's simple but only supports scientific types of loop nest structures. R-Stream [21] can determine the tile size itself which makes it unique. Par4All [26] is an open-

source compilation framework supporting the integrated compilation of applications for GPUs and other heterogeneous platforms. The polyhedral abstraction of its array access makes it a powerful analysis tools. PPCG [30] supports multilevel locality optimizations, multilevel parallelization methods to generate the GPU code of memory and concurrency management.

Another popular target platform for scratchpad research is FPGA. From high level synthesis (HLS) point of view, several works [25][35][7] translate the C source into hardware representation with the automated tilling optimization.

The TI DSP has a more diverse set of on-chip memory structure than CPUs including L1 SRAM, L2 SRAM, and shared memory (MSMC). This unique memory hierarchy makes the mapping between off-chip buffer and on-chip SPM more flexible and gives more optimization room for the kernel. Our preliminary results also indicate how the performance varies when using different combinations under different memory access patterns.

Unfortunately, the current automatic loop tiling schemes do not explore how platform specific SPM configurations affect system performance. This model should represent the capability of the platform such as the bandwidth through each level of memory hierarchy, the data processing ability of ALUs and the cache performance model. These ideas will be illustrated in detail in following chapters.

In summary, the kernels developed using scratchpad method often focus on specialized platforms such Cell and DSP platforms. But there has been limited work on a generalized SPM methodology. Loop tiling tools, on the other hand, are mature on general processors. But not yet adapted specifically for DMA-compatible SPM memories.

Chapter 4

Motivation and Problem Statement

In our preliminary work on SPM mapping and allocation, we hand-tuned a Sparse Matrix Vector Multiplication (SpMV) kernel. Though the main objective is to explore the impact of SPM buffering, we are also noticed how the performance is affected when we applied different SPM mappings to the SpMV kernel data arrays.

In this chapter, we would briefly describe the implementation of this SpMV kernel and show how does the mapping related to the kernel performance.

4.1 The Optimization of SpMV

As described in background chapter, though the SpMV kernel has irregular memory access pattern as **X** and inconsistent consuming rate arrays as **Y** and **ptr**, it is friendly to SPM buffering and loop tiling techniques. The our optimization work is major about how to translate the kernel from a naive version to the SPM buffering version.

Kernel Implementation

The basic implementation of SpMV was a simple, naïve loop that directly performs the kernel as shown in Algorithm 1.

This implementation is straightforward. However, the if-statement inside loop added extra dependency across iterations. With this structure, the compiler has no way to maximize the usage of the VLIW function units. The way to solve this is by introducing the **loop fission**. It splits the single loop body into two. In this

Algorithm 1 Naïve Implementation

Input: $val, col, ptr, y, x, \alpha, \beta$

1: $row \leftarrow initial\_row$
2: **for** $i = coreNum \times (M/cores) \rightarrow (coreNum + 1) \times (M/cores) - 1$ **do**
3: **if** $ptr\,[row] == i$ **then**
4: $row \leftarrow row + 1$
5: $y\,[row] \leftarrow y\,[row] \times \beta$
6: **end if**
7: $y\,[row] \leftarrow y\,[row] + \alpha \times val\,[i] \times x\,[col\,[i]]$
8: **end for**

way, we could make two loops, one with control-independent calculations and one without named the **product loop** (line 1 to 3) and **accumulate loop** (line 5 to 25) respectively in Algorithm 2. The **product loop** has no dependencies and can be fully optimized by the compiler. By removing the multiplies out of the **accumulation loop**, the performance also benefits from reduced register usage and increased data locality. The only side effect of this method is another on-chip SPM buffer is required to hold the intermediate results passed from the **product loop** to the **accumulation loop**.

Mapping to SPM Buffering Implementation

In Algorithm 2, there are five data arrays. The **val** and **col** arrays have a constant consumption rate in the product loop. So, we double buffer them with on-chip SPM. While the **Y** and **ptr** are consuming data at a different rate, they are mapped to a circular buffer. Since **prod** does not need to be written back to memory, we simply map it to an on-chip buffer. The whole mapping picture is shown in Figure 4.1.

Allocation Parameters

Our SpMV kernel is parameterizable, allowing each buffer to be sized and allocated at runtime using a given configuration. In order to evaluate the impact of allocation decisions, we ran the kernel with a full enumeration of valid buffer mappings.

Algorithm 2 Loop Fission

Input: $val, col, ptr, y, x, \alpha, \beta$

1: **for** $i = 0 \rightarrow M$ **do** //product loop
2: $prod\,[i] \leftarrow \alpha \times val\,[i] \times x\,[col\,[i]]$
3: **end for**
4: $Acc \leftarrow 0$
5: **for** $i = 0 \rightarrow M$ step by K **do** //accumulation loop
6: $Acc \leftarrow Acc + prod\,[i]$
7: **if** $ptr\,[row] == i$ **then**
8: $row \leftarrow row + 1$
9: $y\,[row] \leftarrow y\,[row] \times \beta + Acc$
10: $Acc \leftarrow 0$
11: **end if**
12: $Acc \leftarrow Acc + prod\,[i + 1]$
13: **if** $ptr\,[row] == i + 1$ **then**
14: $row \leftarrow row + 1$
15: $y\,[row] \leftarrow y\,[row] \times \beta + Acc$
16: $Acc \leftarrow 0$
17: **end if**
18: ...
19: $Acc \leftarrow Acc + prod\,[i + K]$
20: **if** $ptr\,[row] == i + K$ **then**
21: $row \leftarrow row + 1$
22: $y\,[row] \leftarrow y\,[row] \times \beta + Acc$
23: $Acc \leftarrow 0$
24: **end if**
25: **end for**

In this test, more than 800 combinations of SPM mappings and tile size are tested. The input dataset is the tridiagonal matrix. In Table 4.1, we lists top five best mappings, from which we could tell how the mapping affect performance. Table 4.2 lists the several configurations that emphasizes the impact of using the cache instead of the SPM. The results are normalized to the pure cache mapping which serves as the baseline. We found that the pure SPM implementation achieves about 50% speedup compared with the cache for this kernel.

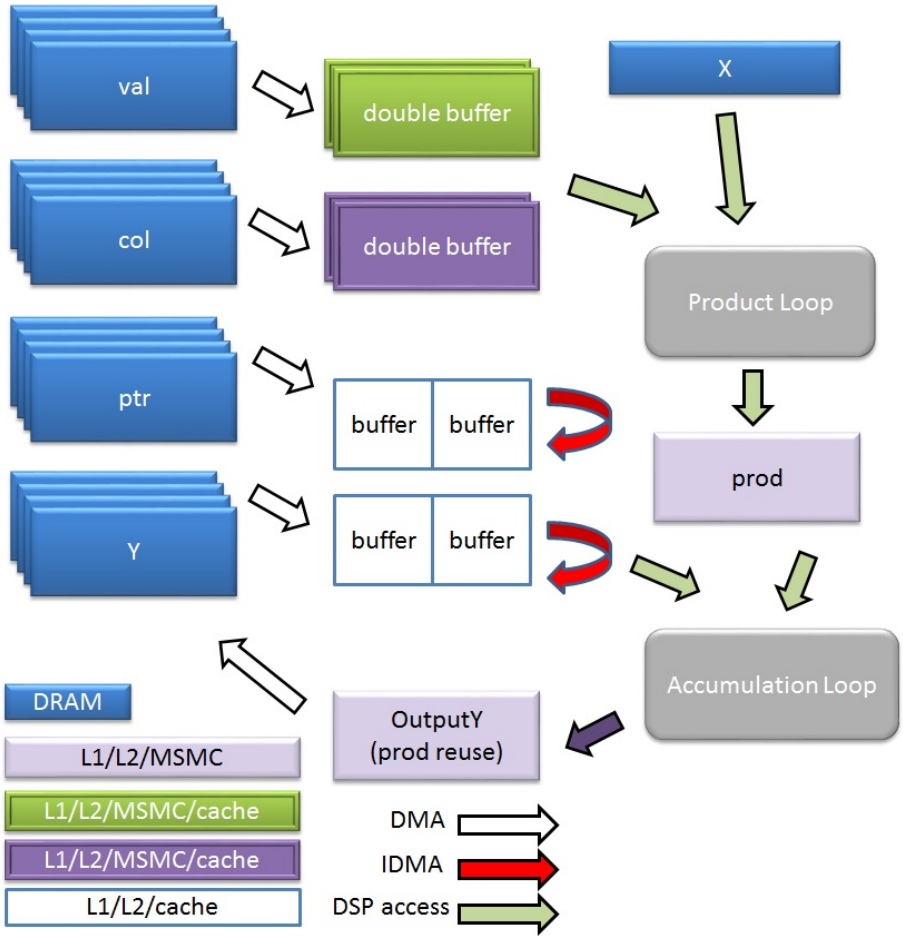

Figure 4.1: Memory system usage.

Table 4.1: Tile size, buffer location and performance

Nonzeroes per row	**Tile size** (bytes)	**val**	**col**	**ptr**	**y**	**prod**	**Gflops**
3	16384	L2	L2	L2	L2	L1	2.26
	16384	S	L2	L2	L2	L1	2.26
	16384	L2	S	L2	L2	L1	2.25
	16384	S	S	L2	L2	L1	2.24
	8192	L2	L2	L2	L2	L1	2.20

L1:level 1 SPRAM, L2:level 2 SPRAM, S:MSMC, C:cache

4.2 Motivation

From the previous experiment, we found that even with the same SPM buffering technique, different mappings lead to significant performance difference. The way

Table 4.2: Cache vs. Scratchpad

Matrix (diagonal)	**Tile size** (bytes)	**val**	**col**	**ptr**	**y**	**prod**	**Normalized Performance**
3	16384	L2	L2	L2	L2	L1	1.49
	16384	L2	C	L2	L2	L1	1.42
	16384	C	L2	L2	L2	L1	1.22
	16384	C	C	L2	L2	L1	1.11
	16384	C	C	C	C	L1	1

L1:level 1 SPRAM, L2:level 2 SPRAM, S:MSMC, C:cache
1: The **col** is mapped to MSMC here, because too many current access will lead to port congestion.

we solve the problem in SpMV is by manually tweaking the code and exhaustively populating all possible mappings. This may not be possible in most application development scenarios.

4.3 Problem Statement

The proposed compiler framework should include the following features.

1. For applicable kernels, it should be able to translate the kernel from a simple and naive version into a loop tiled version.

2. The tool should be able to analyse the kernel and find the optimized mappings between data structure and on-chip SPM buffer. Other than SPM, cached DDR is also under consideration if it can yield better performance.

3. When allocating an array to SPM, it should generate all the supporting code including data load/store to utilize the EDMA engine.

Automatic Compiler Framework

In the problem statement, the first and the third requirements are considered together. The first issue requires adding loop iteration levels. In our framework, we address this issue by **Pluto**. As covered in the related work chapter, the Pluto is a source-to-source translator. It automatically translates the kernel into loop-tiled version with parameters. The Pluto is based on polyhedral analysis on source code.

When using Pluto, the input program is translated into an intermediate polyhedral representation called **cloog**. With this format, the kernel loses all its symbolic information, but keeps the loop structure, data dependency, and memory access pattern. Based on **cloog**, Pluto is able to manipulate the loop by doing polyhedral transformation on it, which changes the loop structure while maintain semantics.

We modified Pluto such that before generation the target C code, we would take its **cloog** intermediate and extend translation support to SPM buffering. The supporting code handles several jobs like creating local buffers, add EDMA load/store utilities, keeping buffer coherency, etc.

The mapping of the SPM is also based on input parameters. These parameters are provided by another tool targeting the second requirement. This tool also takes the intermediate code generated by Pluto as input, then pass the information to a solver, which would make an optimized choice. In this optimization process, trade-offs will be made.

Issues and Trade-offs

The objective of the allocation is to find the best location of each data array in the DSP. The location or mapping could be on-chip SPM or cached DDR. The decision is based on the memory access pattern of each data structure as well as characteristic of the DSP and peripherals like DDR.

Level 1 Memory

The L1 SPM or L1 cache can be accessed within a single cycle. However, because of its limited capacity and because larger tile transactions are more favourable, we assign L1 as cache.

Level 2 Memory

The latency of L2 is greater than L1, but has comparable bandwidth. The drawback of L2 is its EDMA bandwidth. When reading or writing data from/to L2 with the EDMA engine, traffic go through the EMC (External Memory Controller), which is connected to a 2nd tier NoC with half of the DDR bandwidth as shown in Figure 2.3. Therefore if the kernel has relatively higher I/O requirement than the computation, this disadvantage could be an issue.

Multiple Core Shared Memory Controller

The advantage of MSMC is that it is connected to the DDR directly. While the XMC (Extended memory controller) also provides a higher bandwidth than the EMC, which guarantees a higher throughput between Corepac and MSMC. Table 2.2 shows that the performance of MSMC to be close to L2. However it requires two more cycles on a prefetch miss, while L2 doesn't have this penalty.

Main Memory

All input and output arrays are allocated in DDR. Generally, there are two types of DDR access, issued by EDMA or cache. EDMA performs bulk.

Table 2.2 shows that we could tell that the cache miss stall is significantly higher than the L2 and MSMC. when the cache miss combined with prefetch misses, the performance becomes even worse. But the kernel with data structures in cached DDR do not require EDMA transaction at all. According to our experience, mapping one

data structure to DDR sometimes improves performance. This happens when the DDR is overwhelmed by data request and both the cache and prefetch buffer miss rate are low.

Chapter 5

Performance Model

The aim of our model is to select the best mapping from all the possible pairs between the kernel arrays and the SPMs or cached DDR. For each kernel, the number of possible mappings depend on the number of data arrays. Given the mapping and the kernel, our model should estimate the final performance of each mapping and select the one with the least cycles.

5.1 Top Level Methodology

In order to determine if kernel performance can be improved by allocating tiles into SPM, we must characterize a kernel's memory access pattern and build a model that predicts multi-core performance under various allocations.

To solve this problem, we first profile the kernel by running a subset (sample) of the whole iteration space. With the readings from the performance counter and the DSP cache miss stall cycles in Table 2.2, we are able to solve the performance estimation under a single core configuration. Then, this result is applied to a neural network to derive the multi-core performance estimation. We do the single core performance calculation and multi-core extension one time for each mapping. The mapping with least cycles will be elected as the best one for our estimation.

Notice that we measure the performance by counting its cycles with a performance counter. Since in the DSP embedded system, it is the most convenient and accurate way to evaluate kernels.

The cycles spent by DSP is a summation of computation, stalls, EDMA transactions, and other overheads. Our objective is to estimate each. We begin with the single buffer scheme on a single core, then extend it to a more general case.

In this simple case, the total cycles are simply made up by the time spent on EDMA and the DSP Corepac, since the operations on them are serialized when only using single buffer as described in Chapter 2.

EDMA Transmission Time calculation

Given the SPM mapping, the total amount of data $\mathcal{D}_{all}$ should be the product of the number of tiles, the tile size and the number of arrays in SPM. In our simplified single core case, the EDMA bandwidth $\mathcal{B}$ is bounded by theoretical bandwidth of each node, due to lack of resource competition from other cores. We could calculate the EDMA transmission time as $\mathcal{T}_{edma} = \mathcal{D}_{all}/\mathcal{B}$. Notice that this calculation is only valid for this simplified single core when using a single buffer. When extended to multi-core environment, all the overhead such congestion, arbitration and task queueing need to to be considered.

DSP Corepac time

The DSP is statically scheduled. Because there is no out-of-order execution. The number of cycles per iteration without memory stalls can be determined at compiler time. Let $\mathcal{T}_{pure}$ = **pure computation time**, the theoretical number of cycles needed by the DSP to perform the kernel computation. This value is usually reported by the compiler as part of its loop optimization output, i.e. the software pipeline report. Notice that the L1 memory (cache or SPM) are of the same frequency as the DSP Corepac without latency. If all of the data structure referenced by the kernel are

placed on the L1 SRAM, the actual cycles spent would equal the compiler generated results.

Methods to Measure the DSP stalls

To count the number of memory stall cycles, we profile the kernel. Unfortunately, the current generation of TI DSP does not support the cache profiling performance counter in the DSP Corepac. Assume no L2 cache, L1 cache misses will be forwarded to the XMC prefetch buffer or L2 SPM if the buffer is located in L2 SPM. By measuring the prefetch hit/miss, we are able to calculate the total L1 cache misses. We then calculate the stall cycles using Table 2.2.

Kernel Profiling

Data Arrays Separation

The prefetch buffer only contains one set of counters and is not able to track the behaviour of each data array. To measure the cache performance of each data array, we need to separate it by mapping to the SPMs that are subjected to prefetch buffers. Other arrays are mapped to L2 to avoid prefetch. For example, given three different arrays **A**, **B** and **C**, we measure the profiling results of **A** by mapping it to MSMC while others are mapped to L2. In this way, only cache misses of **A** will be counted by the prefetch buffer counter. The profiling results of **B** and **C** are obtained in a similar way.

Sampled Profiling Method

In order to compute the memory performance of the kernel, we launched the profiling run for three times which is equal to the total number of arrays. If the running instance involves a large dataset, the time spent on profiling could be a significant

issue and make it impractical as a compiler framework. The way we solve this problem is by introducing the **sample run**. It means instead of covering the whole iteration space, we choose to cover a subset.

For instance, the basic operation of the kernel **ssyrk** from LAPACK is to calculate the production of matrix A and its transpose $A^{'}$ B.2. This would require a $O(n^3)$ in time complexity. To profile this kernel, our compiler framework will generate code to process a subset of the tiles, as shown in Figure 5.1.

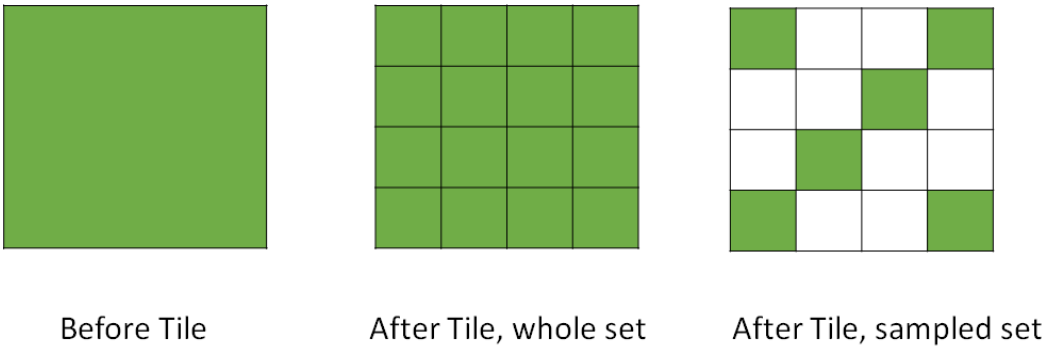

Figure 5.1: The Sample of the Whole Iteration Space after Tile

Our experiment shows in terms of the prefetch buffer performance, the sample run is close to a full set coverage. In Figure 5.2, the DSP version of kernel **ssyrk** is processed with a sample rate from 10 to 100. We could tell that the normalized prefetch hit and miss matches quite well with the full set run(sample rate is 0 in the figure). The normalized results is the production of the actual prefetch buffer reads and the sample rates. Notice that some specific sample rate may not work well. As shown in Figure 5.2, the experiment with sample rate 50 suffers from a surge of prefetch buffer accesses. This is caused by L1 cache thrash, since the memory access stride happens to be integral times of the cache set size.

With the results collected by the separated profillings and sample run, we could correlate the cache behaviour like cache miss rate to each array. The cache behaviour should only related to its memory access pattern rather than the buffer location. For instance, for **A**, regardless of whether it is mapped to L2 or MSMC, the L1 cache

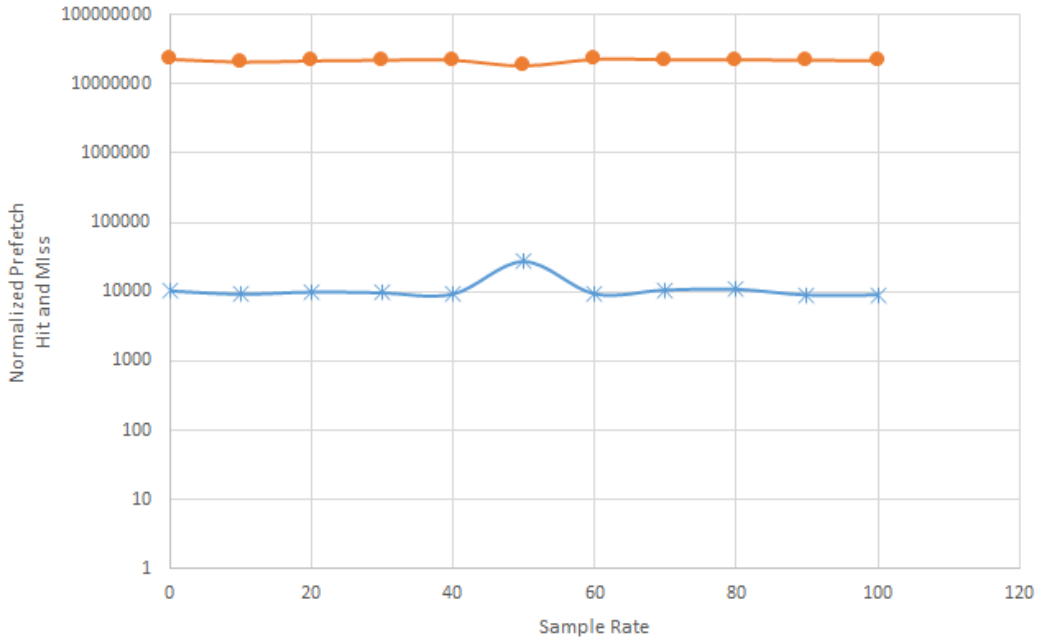

Figure 5.2: DSP Sample Run Results on **ssyrk**. All the results are extrapolated by its sample rate. The dot line is for prefetch miss, the other is hit.

miss rate should be the same. The only difference is the access latency described in Table 2.2.

Performance Correction of the Cached DDR

The data in the Table 2.3 are based on the assumption of sufficient memory bandwidth or no other instances competing for memory bandwidth. However, typically, the traffic from other cores or EDMA will compete for access to DDR. In our model, the extra penalty will be added on top of the prefetch hit/miss latency cycles shown in Table 2.3. As a result, a correction mechanism is necessary.

The two major type competitors for a single DSP core include its peers and the EDMA controller. We measured both scenarios separately.

Multiple-core Performance Penalty

To characterize the effective DDR bandwidth under varying degrees of contentions, we developed a series of micro-benchmarks which are shown in Appendix A.1, A.2 and A.3. These kernels accumulate arrays, but exhibit varying ratios between memory access and computation. By covering all of them, we are able to measure the multi-core effect to kernels with different computation intensity. To compute the final result, they need to read DDR using cache. We scale up the concurrent core numbers from 1 to 8 as in Figure 5.3. With all three functions, we found that the trend of accumulating overhead from peer cores is increasing linearly. The slope of each line in the figure is close to each other, which is about 1.2x to 1.5x. We use this ratio as our first DDR access correction factor α.

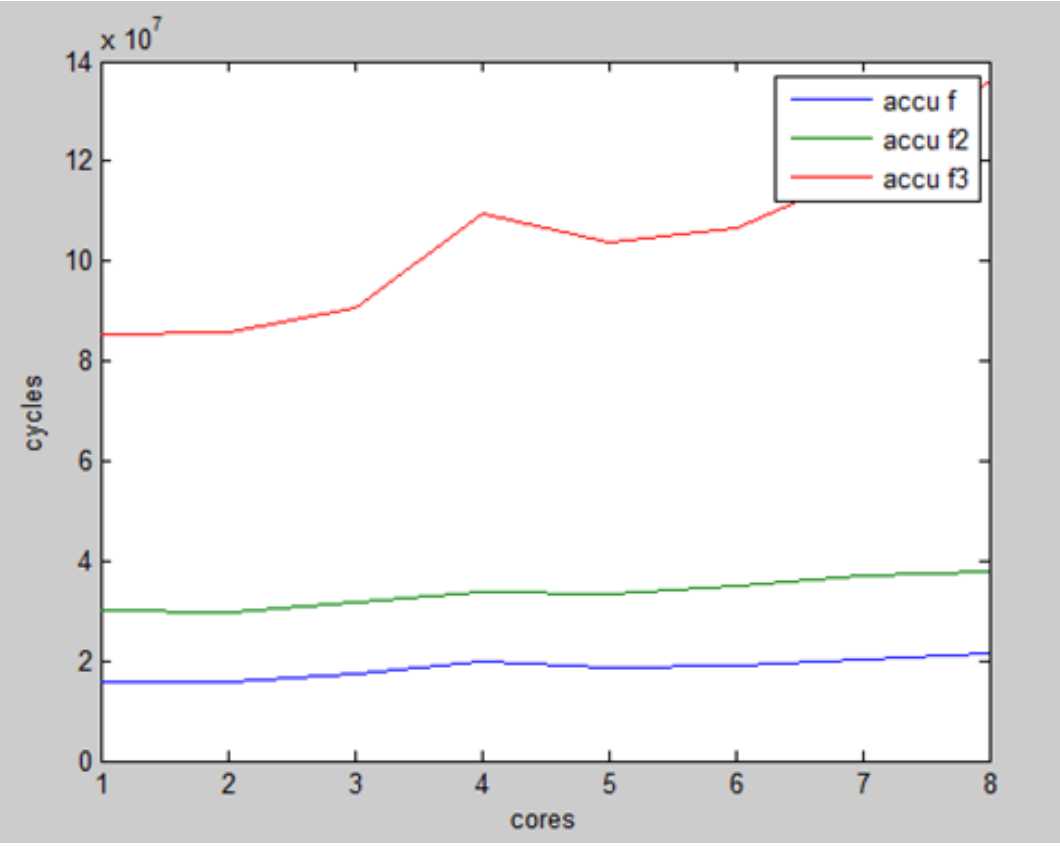

Figure 5.3: Performance on Testbeds when Number of Cores Scale up

Performance Correction for Cached DDR with Concurrent EDMA

Besides the multi-core contest, we also measure the adjusted performance results when the EDMA controllers are also competitors for the DRAM. Every time before we run the functions as shown in Appendix A.1, A.2 and A.3, a number of EDMA transactions will be launched first. The two stages are serialized in the same loop iteration. The loop continues for hundred times over 8 cores to make sure a complete competition amongst cores and the DMA controllers. This micro-benchmarks is shown in A.4.

The functions transfer $\mathcal{S}_a$ bytes from DDR to L2. If we invoke the function m times, the total data size is $m * \mathcal{S}_a$ bytes. Similarly, one launch of the EDMA transaction will transfer $\mathcal{S}_b$ bytes. And if the EDMA is launched for n times, the data size moved will be $n * \mathcal{S}_b$.

In our experiment, by adjusting the number of m and n, we are able to adjust the total DDR access ratio made by cache and EDMA from $0:1$ to $20:1$. Because we are only interested in the cached DDR performance, by subtracting the pure computation time and the EDMA time to get the time spent on the cached DDR access time $\mathcal{T}_{ddr}$, we generate the results shown in Figure 5.4.

We could easily tell when $m:n$ less than 2, there is a performance increment for $\mathcal{T}_{ddr}$. While after that, the function cycles stays, which indicates more EDMA traffic would have no effect to the time spent on DDR by cache. The interesting thing is when $m:n$ is less than 2, the cache performance increased while the EDMA transactions are mixed in. Actually this does not mean the cached DDR performance increment as it appears to do. Instead, it implies the EDMA transactions are somehow handled by the DDR controller in between its service to cache. In other words, even when the cache is constantly servicing misses, it could not make the DDR controller fully utilize its memory bandwidth, and when some amount of EDMA transactions get involved, they could take use of the idle stages of the memory controller and improve

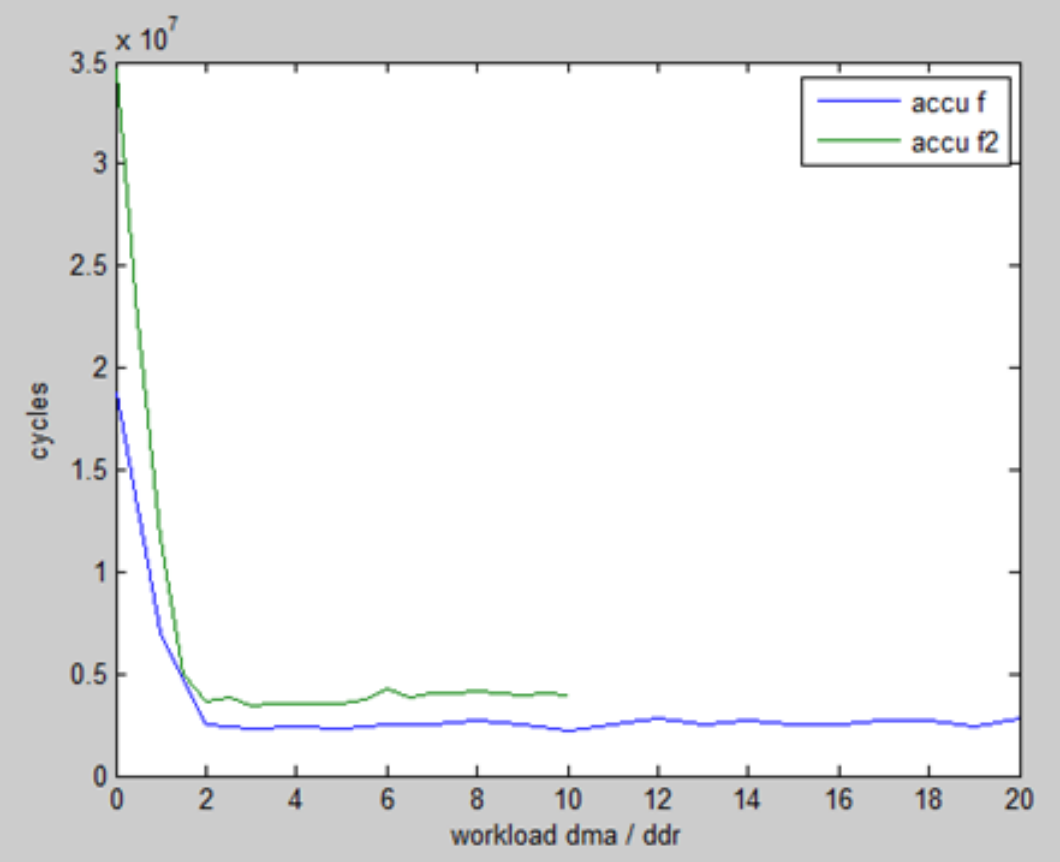

Figure 5.4: Performance on Testbeds when Number of Cores Scale up

performance.

To further explore what happens when $m : n$ less than 2, we make this ratio more fine grain by make changing it from 0 to 2 with a 0.1 increment every time as shown in Figure 5.5. Obviously, the performance decrement of cache is linear to increment of the EDMA transmitted bytes.

When calculating the cached DDR latency, if there is any EDMA transaction ongoing in other peer cores, we should decrease the access latency with $\beta(r)$ from the Table 2.2, r is the ratio between cache transmitted bytes and EDMA transmitted bytes.

Notice that these rules may not apply to the MSMC access, since the bandwidth between the MSMC and each DSP core is 19.2 GB/s. There is quite enough of bandwidth compared to the DDR. So except for some extreme situations, the access

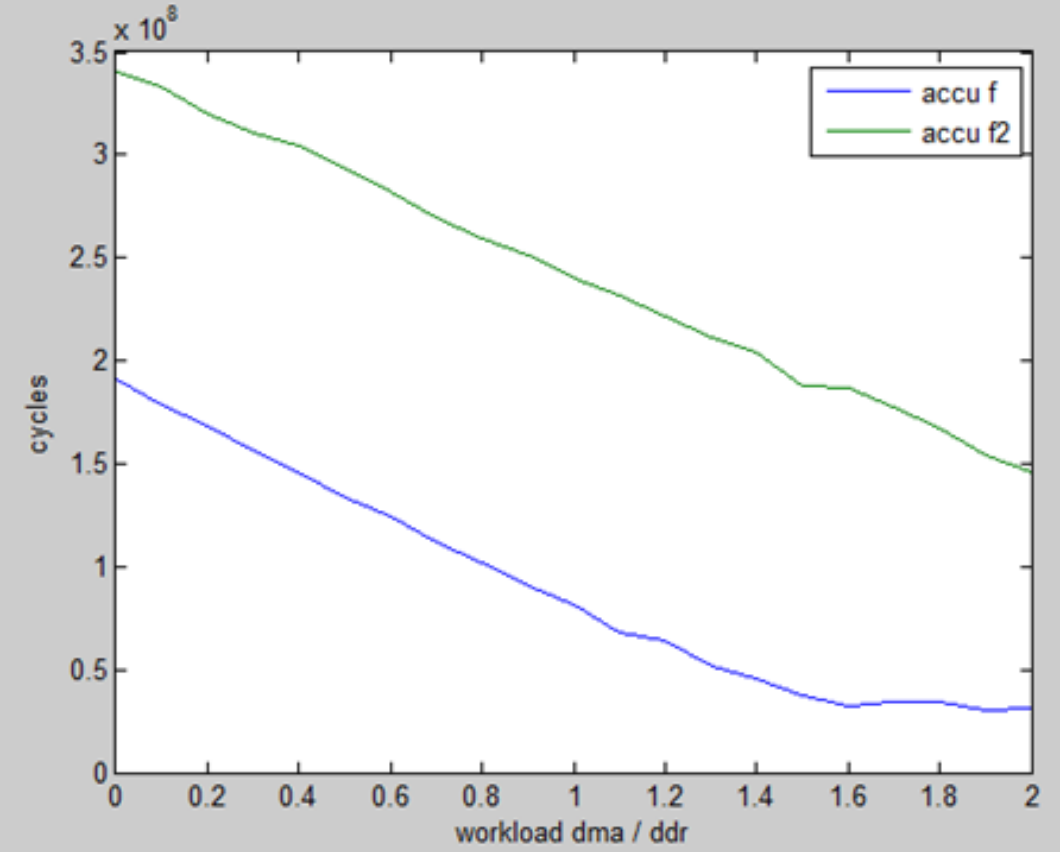

Figure 5.5: Performance on Testbeds when Number of Cores Scale up

latency to MSMC should always honor the values in Table 2.2.

Single Core Performance

The whole equations used to calculate the time is $\mathcal{T}_{all} = \mathcal{T}_{compute} + \mathcal{T}_{edma}$. Notice that, $\mathcal{H}$ and $\mathcal{S}$ are the prefetch hit and miss respectively, while $\mathcal{S}$ and $\mathcal{L}$ are the DSP stalls for prefetch hit and miss respectively. For EDMA, $\mathcal{C}_{tile}$ is the tiled buffer size and $\mathcal{B}$ is the channel bandwidth. I, J and K represent the number of data structures mapped to MSMC, L2 and cached DDR respectively. α and $\beta(r)$ are correction factor to the cached DDR access mentioned in previous section. The equation $\mathcal{T}_{l2} = \sum_j^J (\mathcal{H}_j + \mathcal{M}_j) * \mathcal{S}_{l2}$ to compute the $\mathcal{T}_{l2}$ is different from others. This is because the L2 is inside the DSP Corepac, only access latency is applied when L1 cache misses.

Mapping with best performance on single core does not always lead to same result on multiple core environment. Because with multiple cores, the resources outside the Corepac are not exclusive to the DSP, such as EDMA transaction, cache missed access to DDR are subject to contention for public resource. On the other hand, $\mathcal{T}_{compute}$ in Equation 5.1 only matters the Corepac or DSP core exclusive resource and has no contention on shared DDR and bus. The cycles here should keep the same when escalating to multiple cores. For this issue, we already address the cache missed DDR by our corrections. We take these measures to DDR in as single core execution time rather than multi-core overhead. Because the DDR is special compared with other SPMs, the method we used are also quite different from each other.

$$\mathcal{T}_{all} = \mathcal{T}_{compute} + \mathcal{T}_{edma} \tag{5.1}$$

$$\mathcal{T}_{compute} = \mathcal{T}_{pure} + \mathcal{T}_{msmc} + \mathcal{T}_{l2} + \mathcal{T}_{ddr} \tag{5.2}$$

$$\mathcal{T}_{msmc} = \sum_{i}^{I} (\mathcal{H}_i * \mathcal{S}_{msmc} + \mathcal{M}_i * \mathcal{L}_{msmc}) \tag{5.3}$$

$$\mathcal{T}_{l2} = \sum_{j}^{J} (\mathcal{H}_j + \mathcal{M}_j) * \mathcal{S}_{l2} \tag{5.4}$$

$$\mathcal{T}_{ddr} = \sum_{k}^{K} (\mathcal{H}_k * \mathcal{S}_{ddr} + \mathcal{M}_k * \mathcal{L}_{ddr}) * \alpha * \beta(r) \tag{5.5}$$

$$\mathcal{T}_{edma} = \sum_{m}^{I+J} \frac{\mathcal{C}_{tile}}{\mathcal{B}} \tag{5.6}$$

5.3 Multiple-core EDMA Scalability

The performance impact of DDR contention can be modeled by using data gathered from micro-benchmarks results. We can characterize kernel behaviour by its com-

pute/memory ratio, assuming all cores are perform the same kernel. In other words, given the $\mathcal{T}_{compute}$ and $\mathcal{T}_{edma}$ ratio on single core, there should be only one corresponding multiple core performance yield. Both the times could be estimated based on our knowledge to the kernel and the SPM configurations covered in previous section.

As such, we could create many of synthetic kernels with different ratios of $\mathcal{T}_{compute}$ and $\mathcal{T}_{edma}$. And based on their results, we could do a regression to fit the curve and get the analytical relationship between the two.

Model Construction

We choose a computation kernel as in Appendix A.1 as well as the EDMA transmission. If we place the data set to any of the SPM or DDR except for the L1, the performance would be downgraded for access latencies. In this experiment, in order to achieve deterministic number of compute cycles, all the data structure in the kernel are located in L1 SPM.

The second step is to choose the EDMA transaction size. Suppose the time of one EDMA transaction is $\mathcal{E}$ and the total cycle spend for one pass of the kernel is $\mathcal{C}$. Our aim is to make it flexible to adjust the ratio between $\mathcal{E}$ and $\mathcal{C}$. We choose the size of the EDMA transaction to make $\mathcal{E}$ as close as $\mathcal{C}$. As a result, if we execute the kernel for a times and EDMA transaction b times in one loop iteration, that ratio would become $b * \mathcal{E}/a * \mathcal{C}$. The code is shown in A.5.

In the following step, we sweep a from 0 to 10 and b from 1 to 10. This yields 110 possible cases covering with different ratio of $b * \mathcal{E}/a * \mathcal{C}$. By launching the kernels on all 8 cores and recording results, we take the ratio of $b * \mathcal{E}/a * \mathcal{C}$ as the X-axis, the performance counter recorded cycles as Y-axis, the value is normalized by dividing $b * \mathcal{E} + a * \mathcal{C}$. In this way, we generated the curve to reflect the relationship between single core information and eight core performance with only considering the computing and EDMA time in single core. The curve is shown in Figure 5.6.

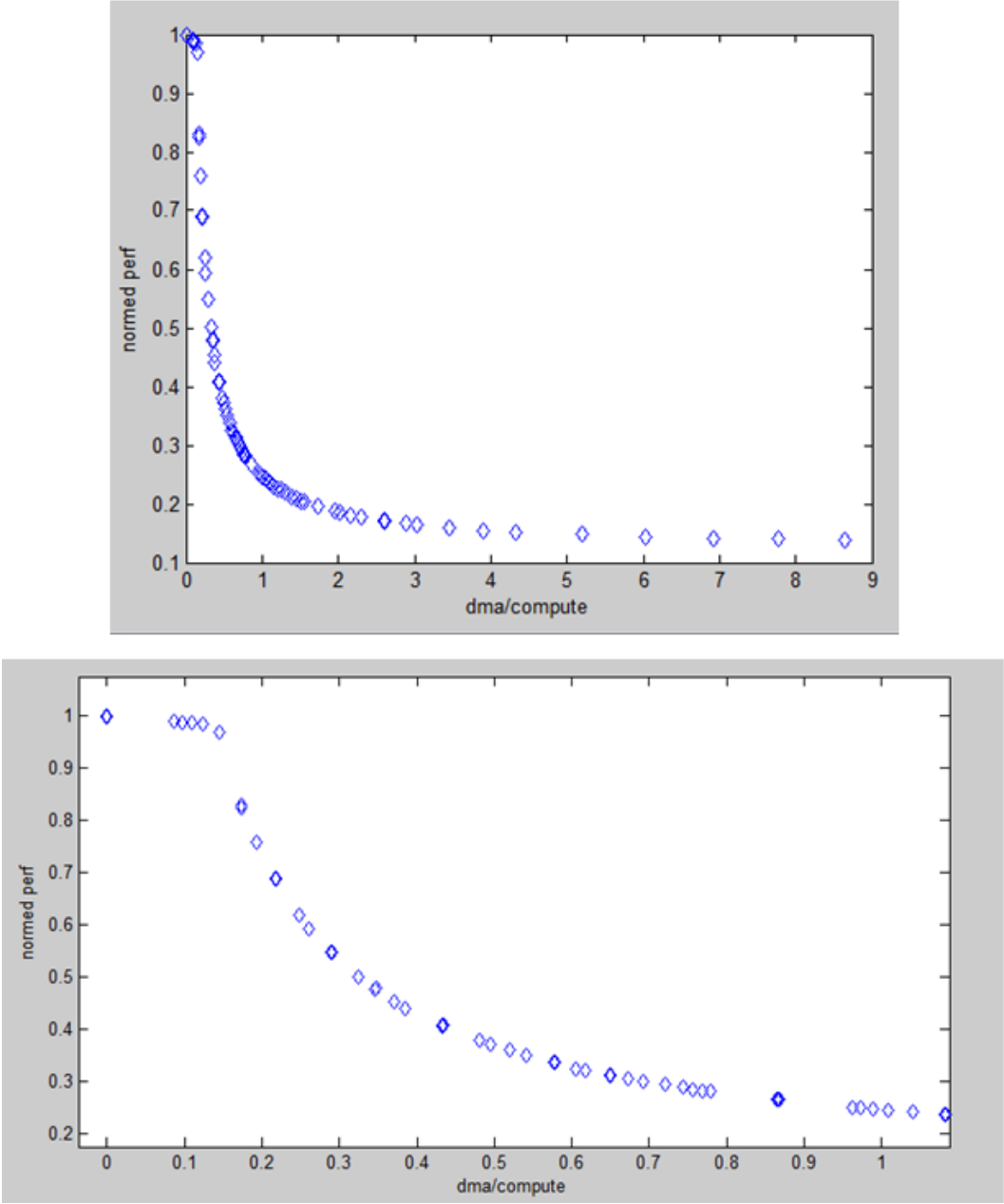

Figure 5.6: Performance on Testbeds when Number of Cores Scale up

Regression with Neural Networks

Once the curve is generated, on any given value from X-axis, we are able to estimate the corresponding performance on the Y-axis. This requires us to represent the curve in an analytical way.

Here, we choose to use Neural Networks to do regression analysis of our model.

Neural networks have a long history to be the tool for function approximation especially when tackling the non-linear problems. Unlike other linearization based method like segmentation and transformation, it require very little knowledge about the curve. And from the curve in Figure 5.6, we could clearly tell this is also a non-linear and an obvious clue about what kind of linearization could be applied.

In a supervised machine learning algorithm. The training data is needed to create the model. And In our previous experiment, we have collected more than a hundred samples to the system covering most application scenarios with DSP computation and EDMA transmission. We use these samplings as the input to train the neural network.

The configuration of the neural network is quite basic. We choose two-level network architecture with hidden layer size of 10 neurons. The training optimization function is the common Levenberg-Marquardt algorithm (LMA) and the cost function is the mean-square-error. During the training, 5% of our data are employed as validation and test respectively, while the remaining 90% are for training.

The measure of our neural network to fit the data is shown in the regression plot, Figure 5.7. This plot shows the actual network outputs plotted in terms of the associated target values. We could find the linear fit to this output-target relationship is closely intersect the bottom-left and top-right corners of the plot. This indicates the network has learned to fit the data well.

With the neural network, all the complexity of the share resource contentions are covered by this model. We are able to derive the estimated performance on eight cores based on the knowledge of the kernel itself and the single core profiling. In this case, given a kernel after profiling run, we could populate all the possible mappings between SPM and data structure and simple pick up the one with least estimated cycles. The detailed procedure is shown in Figure 5.8.

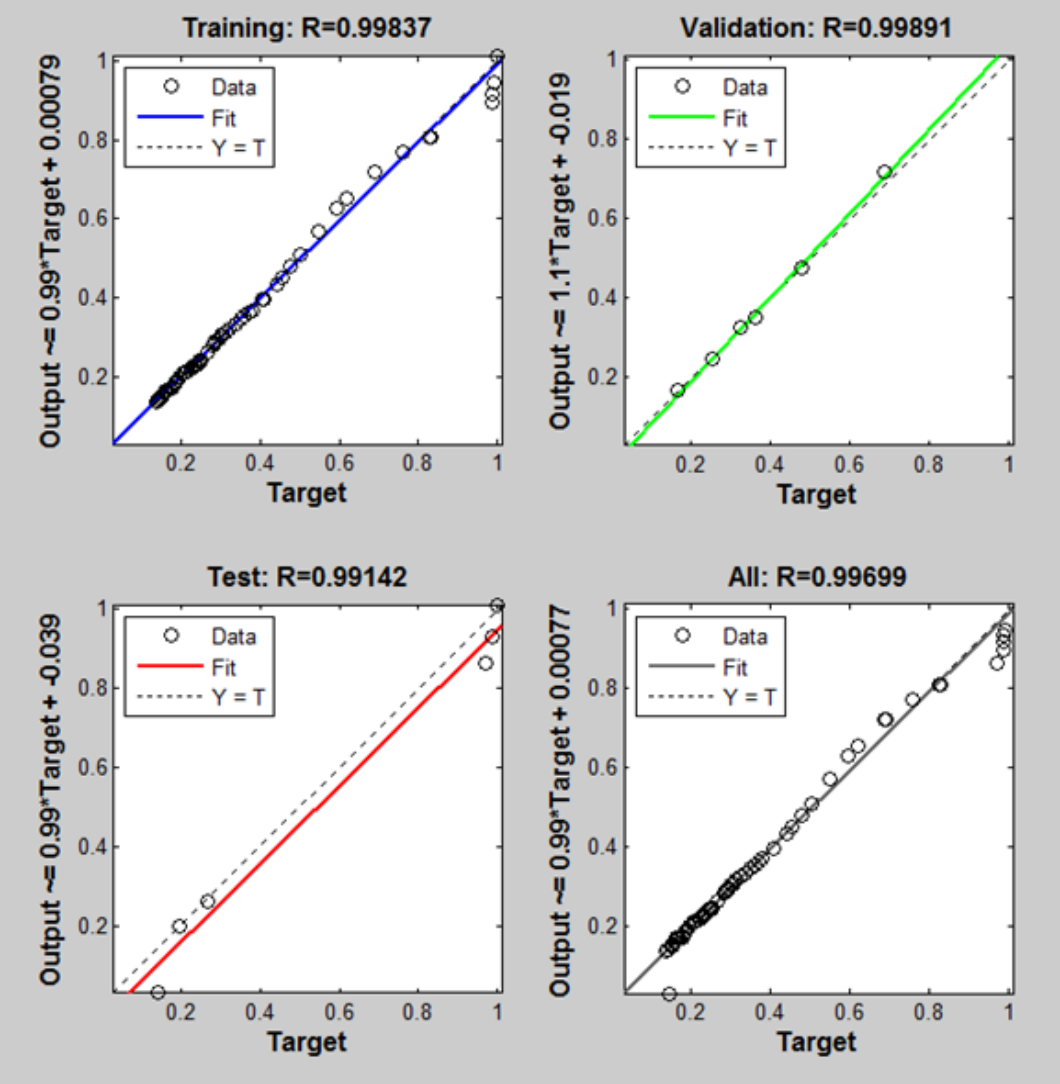

Figure 5.7: The Result of Neuronetwork Regression

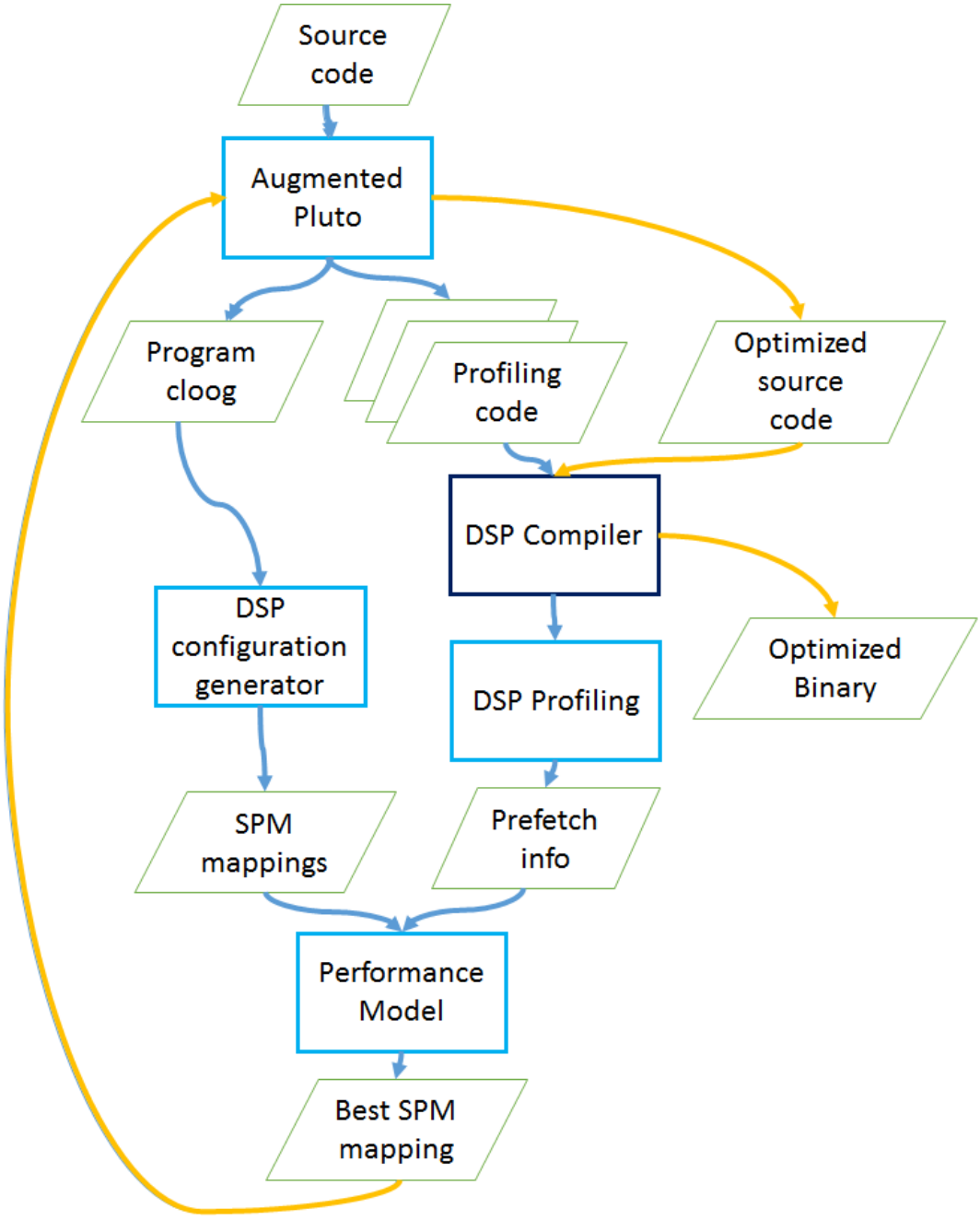

Figure 5.8: The Procedure of Processing

Chapter 6

Model Performance Evaluation

In order to evaluate the effectiveness of our compiler framework and the optimization model, the five most common scientific computation kernels are selected. They differ in computation intensity, dimensions of the loop body, and number of data arrays. In the first chapter, we would like to give a brief introduction to each kernel and summarize their features of them.

6.1 Kernels to Compute

Among the most basic kernels in scientific computing area are the linear algebra and stencil computations. So, we select the matrix multiplication and the **ssyrk**.

• The matrix multiplication is one of the most computing intensive kernel. It's widely adopted by a variety of benchmarks. It has three data structures with dramatically different access pattern from each other.

• **ssyrk** is a level 3 BLAS routine performing symmetric rank k operations. We selected in this kernel also due to its complexity, which is $O(n^3)$ on its iteration space.

For stencil computing kernels, we have both 1-D and 2-D Jacobi kernels.

• The 2-D kernel is more computation intensive than the 1-D kernel. Because, with SPM based loop tiling, the utilization of the data on boarder is much less than the inner-side ones, while the two type of data consume the same amount of EDMA bandwidth. This added a overhead to the loop tiling scheme with small tiles.

• For 1-D stencil, the computation/load ratio is fairly small. This feature make this kernel heavily rely on memory bandwidth. Both its spatial and temporal locality

Table 6.1: Evaluation Kernels

Kernel	Category	Dimention of Interaction Space	Halo-region Redundancy	Data Arrays	Other
2-D Jacobi	Stencil Loop	2	Y	2	Space filtering
9 points Stencil	Stencil Loop	2	Y	2	More points stencil
1-D Jacobi	Stencil Loop	1	Y	2	highly memory-intensive
Dense Matrix Multiplication	Linear Algebra	3	N	3	highly compute-intensive
ssyrk Kernel	Linear Algebra	3	N	2	highly compute-intensive

are high. This decrease the margin of SPM performance advantage over the cache. That's why we also want to include this kernel.

• The last one is the 9 point stencil. It's identical to the 2-D Jacobi, except more points are involved. This would potentially increase the computation intensity on this 2-D kernel. The characteristic of each Kernel are summarized in Table 6.1. The source code of loop tiled version for each kernel are listed in B.1, B.2, B.3, B.4 and B.5. This code is generated by our compiler tools.

6.2 Performance Results

Ground Truth

The first step of our experiment is to collect the ground truth. This involves a full enumeration of all the possible mappings to every kernel. Since our compiler framework supports parameter controlled SPM mapping code generation, we could

Table 6.2: Top 3 Best Mapping, Ground Truth and Model Results

Ground Truth				
		1st Performance	2nd Performance	3rd Performance
Matrix Multiplication	A	L2	L2	MSMC
	B	L2	MSMC	MSMC
	C	L2	L2	L2
ssyrk	A	L2	MSMC	L2
	C	L2	L2	MSMC
2-D Jacobi	A	L2	MSMC	L2
	C	MSMC	MSMC	L2
9 points stencil	A	MSMC	L2	MSMC
	C	L2	MSMC	MSMC
1-D Jacobi	A	MSMC	MSMC	MSMC
	C	DDR	MSMC	L2
Model Results				
Matrix Multiplication	A	L2	L2	L2
	B	L2	MSMC	L2
	C	L2	L2	MSMC
ssyrk	A	L2	L2	MSMC
	C	L2	MSMC	L2
2-D Jacobi	A	L2	MSMC	L2
	C	MSMC	MSMC	L2
9 points stencil	A	L2	MSMC	L2
	C	MSMC	MSMC	L2
1-D Jacobi	A	MSMC	MSMC	MSMC
	C	DDR	MSMC	L2

easily generate all the kernels by choosing different configuration file as input. Then we compile and run these kernels on our TI Keystone II DSP platform which has been elaborated in chapter 2. After we covered all the testing kernels, We simply pick up the top 3 mappings for each kernel listed in Table 6.2.

The matrix multiplication and the ssyrk are kernels with highest complexity and are computing intensive. In the Table 6.2, we could find the L2 SPM is more favourable for both of them. This is due to its low access latency when comparing with MSMC and cached DDR. In other hand, the advantage of MSMC is its high EDMA transmission bandwidth. This help the configuration with MSMC outperforming others in kernel 2-D Jacobi and 9 point stencil. The 1-D Jacobi is a heavily

memory bounded kernel and its cache miss rate is very low due to its high spacial locality. In our previous analysis, we already proved companied with other EDMA transactions we could see a performance boost of cached DDR. This is also proved in this kernel. The 2nd place of that kernel is dual MSMC configuration, which is also caused by the high memory bandwidth requirement.

Model Testing Results

To evaluate our performance model as well as the compiler framework, we take the untiled cache version of each kernel and feed it to the procedure as in Fig 5.8. Each kernel is profiled with a sample rate 100 to get the prefetch buffer hit/miss. The model generated all the possible mappings ranked by least cycle as in Table 6.2. We could find that except for the 9 point stencil, the prediction of the best performance mapping made by the model matches the real case. And for 9 point stencil, the best mapping given by the model is on the 2nd place in the ground truth table 6.2. These results proves our model is successful to elect best performance mapping.

Speed-up over Cache

One of our major contributions is to automate the SPM buffering by our compiler framework. To show how much we gain from the cached-only version of the code, we normalized the results from each mapping with the total cycles of cached-only version. In Figure $6.1 - 6.5$, the mappings are represented by numbers, for instance, the best mapping of Matrix Multiplication is **222**. The three digits, each indicates a mapping for a single data array in the kernel. In Matrix Multiplication B.1, there are three data arrays **A**, **B** and **C**. This means, if all three of them are mapped to L2 SPM, the performance is the best, as we showed in Table 6.2. In this way, the quantitative results are also demonstrated. Notice that, the mappings are ranked by the performance of ground truth. Besides the ground truth, our estimated results all

listed by side. The performance difference between the two is shown by line chart with 2nd Y-axis. It is normalized by the ground truth and shown in percentage.

For Matrix Multiplication and ssyrk, the ground truth mapping achieved a 3.6x speed-up over the cache version. Though our model overestimates the performance, the trend from mapping to mapping follows the ground truth quite closely, except for the mapping **23** in ssyrk. In the 2-d Jacobi and 9 point stencil tests, our model also did a good job to predict the top mappings ranks. However, the variance in terms of speed-up across mapping are not as obvious as the ground truth. This may be caused by the assumption we made to simply the cache behaviour when DSP writing to memory. Due to the existing of write buffer, we assume no extra penalty for write. For these two kernels, the ratio between read and write are low comparing with the first two linear algebra kernels. The estimation of 1-d Jacobi kernels behaves well. Its read/write ratio is similar with 2-d Jacobi, but with much less cache miss. Also, after our corrections, the value of cached DDR comes up, which matches the ground truth quite well.

From quantitative results, we could find that by simply applying SPM buffering technique to the naive code without any code optimization, we can achieve at least a 3.5x speed-up which automatic with our compiler framework. The performance model helps to locate the best mapping with high accuracy although the performance variance across mappings are not reflected quite well in some of the kernels.

6.3 Timing Results

To find our ground truth, we enumerated every mappings within their whole iteration space. This is too long to be an optimization step. However, with our sampling scheme and performance model, the time spent has been dramatically decreased to a normal compiling procedure level as shown in Table 6.3.

In this table, we analysed the time consumption by comparing the Ground Truth

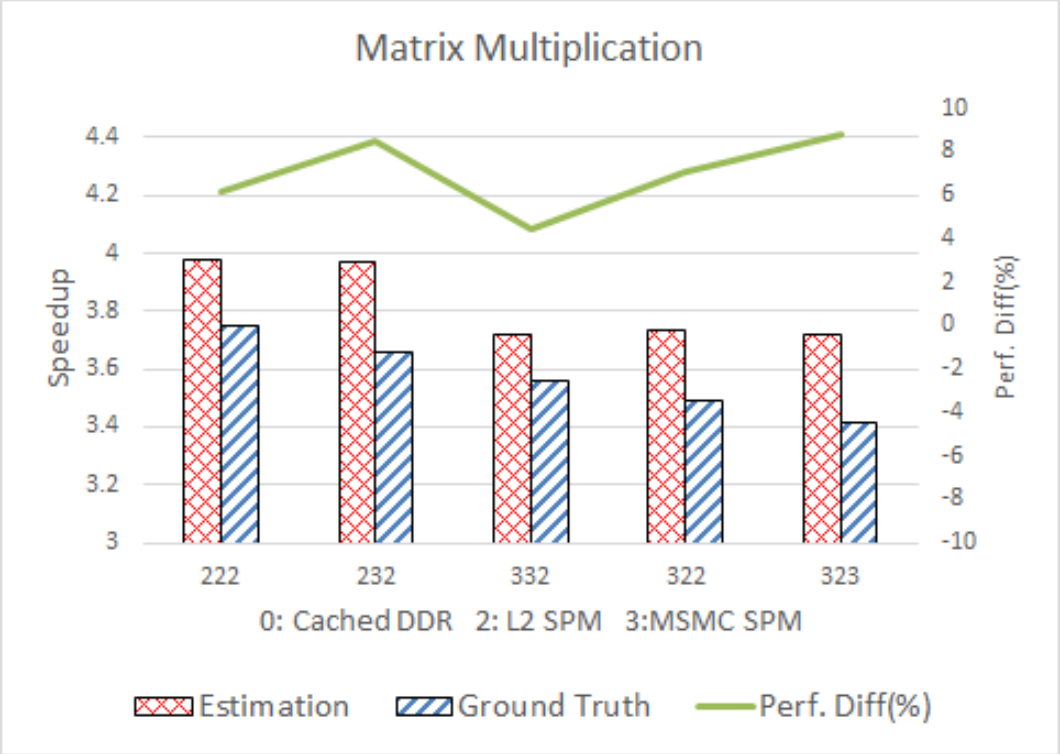

Figure 6.1: The Result of Matrix Multiplication

and the performance model. For the compile, launch and run time, the enumeration needs to cover every single mapping in a whole set run. While in our model, we only need to launch the kernel one time for each array for sample run. The only disadvantage of the model is its time for optimization, which also requires enumeration all the mappings. However, since the neural network are pre-generated, and real time spent by model are trivial when comparing t and r in the table.

In this matrix multiplication example, the model prediction method is several hundred times faster than the enumeration method. For those kernels with two arrays, the search area is only one third to matrix multiplication. Even though, our method is still able to achieve a hundred times speed-up.

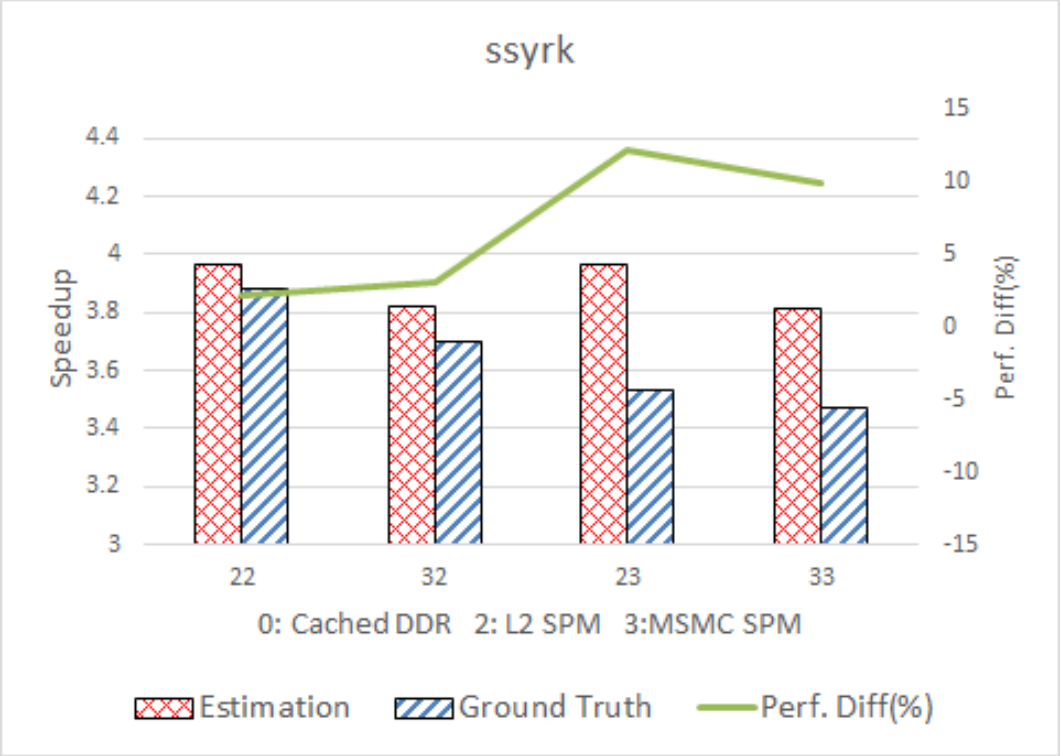

Figure 6.2: The Result of ssyrk

Table 6.3: Time spent on Getting Ground Truth or Model prediction

Matrix Multiplication		
	Ground Truth	Our Model
SPM Options	3	3
Number of Arrays	3	3
Sampling Rate	0	100
Compile and Launch Time (t^1)	$3^3 * t$	$3 * t$
Run Time (r^1)	$3^3 * r$	$3 * r/100$
Optimization (o^1)	0	$3^3 * o$
Total	$3^3 * t + 3^3 * r$	$3 * t + 3 * r/100 + 3^3 * o$

1: **t**, **r**, and **o** are corresponding processing time for single kernel.

6.4 Summary and Other Issues

In this chapter, we evaluated both our compiler framework and the performance model. For our compiler, it works quite well to automatically generate code for loop tiled kernels with SPM. The speed-up are easily achieved with no extra effort from the programmer. The kernels we selected are all supported by SPM quite well. However,

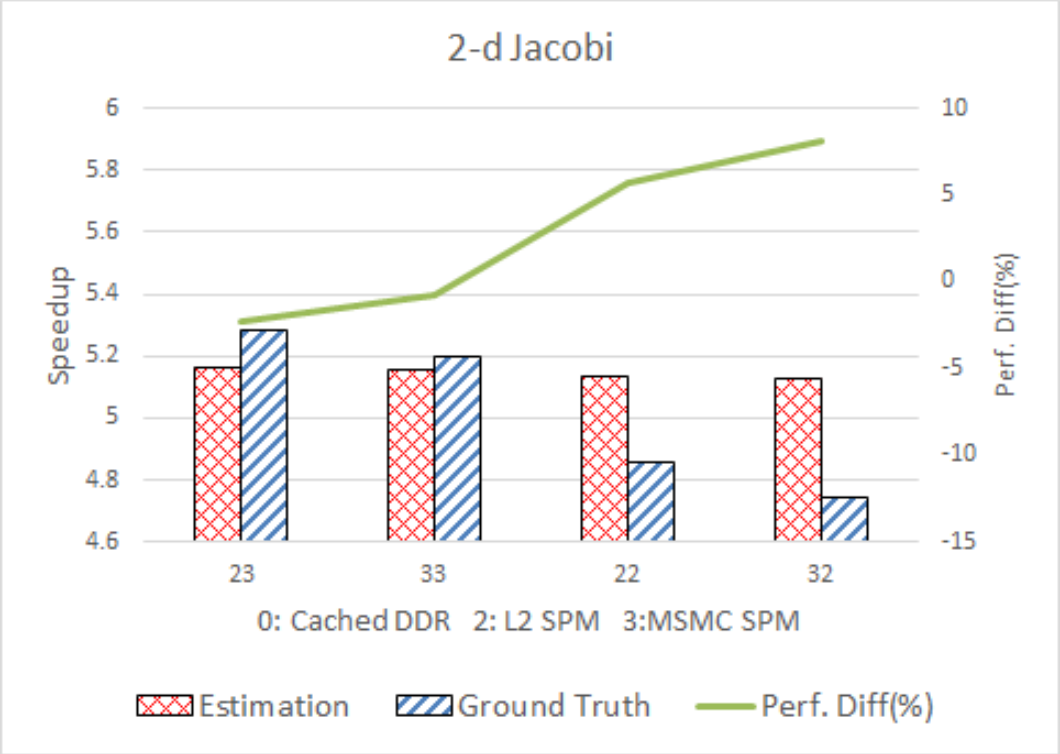

Figure 6.3: The Result of 2d-Jacobi

they are only a subset of the wide range scientific computing kernels. In the future, our model should support to identify the kernels that are not tilable and simply return the input code.

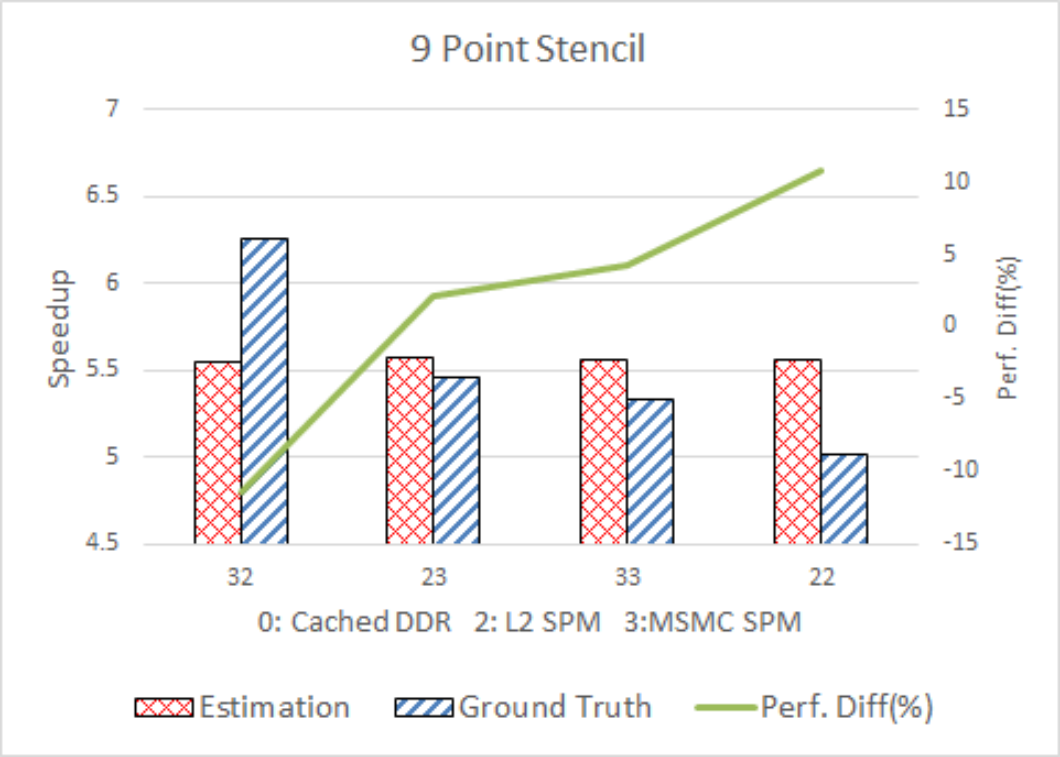

Figure 6.4: The Result of 9 Point Stencil

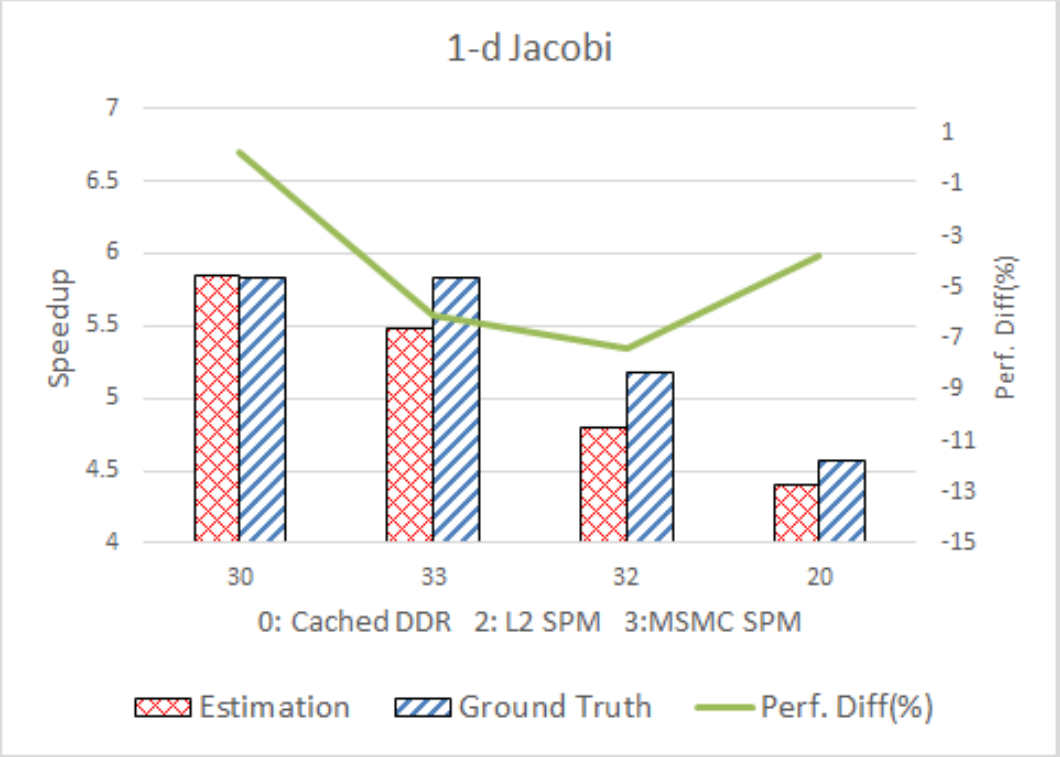

Figure 6.5: The Result of 1d-Jacobi

Chapter 7

Conclusion

Both loop tiling and SPM buffering technique have long been proved their effectiveness in optimizing computing kernel. However, most of the previous work has to be deployed manually by the programmer due to lacking an automated compiler framework. This procedure could be complex and error-prone. But the more important point is that the kernel may not get expected performance boost due to improper SPM mapping. And in our previous work on SpMV, we proved this is vital to performance.

In this dissertation, we augmented the existing loop tiling compiler frame work Pluto to support SPM buffering. To solve the optimization problem, we describe a novel performance model to decide the best mapping between data arrays and the on-chip SPMs. This model is built based on the analysis of hardware characteristics and the SPM buffering techniques. Working together with our compiler framework, this could help a programmer to quickly translate the computing kernel into loop tiled version with SPM support.

We evaluated our model and framework on several scientific computing kernels. In most cases, our model could accurately identify the best mapping out of the whole enumeration space because it covers the factors which could affect the single core performance. When extended to multiple cores, the neural network helps to address the contentions on shared DDR and bus. All these optimization happen automatically during compiling. The whole procedure only takes a reasonable time compared with the way to obtain the ground truth.

Our current work is based on standalone processing tools as in Figure 5.8 and

fulfilled by script. In the future, this work could be integrated into the TI OpenCL environment to help the computing kernel code generation with best hardware utilization and performance.

Bibliography

[1] Murtaza Ali, Eric Stotzer, Francisco D Igual, and Robert A van de Geijn, *Level-3 blas on the ti c6678 multi-core dsp*, Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on, IEEE, 2012, pp. 179–186.

[2] Abdel-Hameed Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng, *The efficacy of software prefetching and locality optimizations on future memory systems*, Journal of Instruction-Level Parallelism **6** (2004), no. 7.

[3] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M Balakrishnan, and Peter Marwedel, *Scratchpad memory: design alternative for cache on-chip memory in embedded systems*, Proceedings of the tenth international symposium on Hardware/software codesign, ACM, 2002, pp. 73–78.

[4] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan, *Automatic c-to-cuda code generation for affine programs*, Compiler Construction, Springer, 2010, pp. 244–263.

[5] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan, *A practical automatic polyhedral parallelizer and locality optimizer*, ACM SIGPLAN Notices **43** (2008), no. 6, 101–113.

[6] Siddhartha Chatterjee, Erin Parker, Philip J Hanlon, and Alvin R Lebeck, *Exact analysis of the cache behavior of nested loops*, vol. 36, ACM, 2001.

[7] Alain Darte, Alexandre Isoard, et al., *Parametric tiling with inter-tile data reuse*, IMPACT 2014 (2014).

[8] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick, *Optimization and performance modeling of stencil computations on modern microprocessors*, SIAM review **51** (2009), no. 1, 129–159.

[9] Paul Feautrier, *Some efficient solutions to the affine scheduling problem. i. one-dimensional time*, International journal of parallel programming **21** (1992), no. 5, 313–347.

[10] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M Mendias, *An integrated hardware/software approach for run-time scratchpad management*, Proceedings of the 41st annual Design Automation Conference, ACM, 2004, pp. 238–243.

[11] Yang Gao and Jason D Bakos, *Sparse matrix-vector multiply on the texas instruments c6678 digital signal processor*, Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on, IEEE, 2013, pp. 168–174.

[12] Carlos García, Francisco D Igual, Guillermo Botella, Manuel Prieto, and Francisco Tirado, *Non-negative matrix factorization on low-power architectures: a comparative study*, Proceedings of the 20th European MPI Users' Group Meeting, ACM, 2013, pp. 175–178.

[13] Somnath Ghosh, Sharad Malik, and Margaret Martonosi, *Cache miss equations: a compiler framework for analyzing and tuning memory behavior*, ACM Transactions on Programming Languages and Systems (TOPLAS) **21** (1999), no. 4, 703–746.

[14] Somnath Ghosh, Margaret Martonosi, and Sharad Malik, *Cache miss equations: An analytical representation of cache misses*, Proceedings of the 11th international conference on Supercomputing, ACM, 1997, pp. 317–324.

[15] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm, *The influence of processor architecture on the design and the results of wcet tools*, Proceedings of the IEEE **91** (2003), no. 7, 1038–1054.

[16] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick, *Implicit and explicit optimizations for stencil computations*, Proceedings of the 2006 workshop on Memory system performance and correctness, ACM, 2006, pp. 51–60.

[17] Jakub Kurzak, Wesley Alvaro, and Jack Dongarra, *Optimizing matrix multiplication for a short-vector simd architecture–cell processor*, Parallel Computing **35** (2009), no. 3, 138–150.

[18] Jakub Kurzak, Alfredo Buttari, and Jack Dongarra, *Solving systems of linear equations on the cell processor using cholesky factorization*, Parallel and Distributed Systems, IEEE Transactions on **19** (2008), no. 9, 1175–1186.

[19] Jakub Kurzak and Jack Dongarra, *Implementation of mixed precision in solving systems of linear equations on the cell processor*, Concurrency and Computation: Practice and Experience **19** (2007), no. 10, 1371–1385.

[20] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M Badia, *Scheduling dense linear algebra operations on multicore processors*, Concurrency and Computation: Practice and Experience **22** (2010), no. 1, 15–44.

[21] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin, *A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction*, Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, ACM, 2010, pp. 51–61.

[22] Amy W Lim, Gerald I Cheong, and Monica S Lam, *An affine partitioning algorithm to maximize parallelism and minimize communication*, Proceedings of the 13th international conference on Supercomputing, ACM, 1999, pp. 228–237.

[23] Amy W Lim, Shih-Wei Liao, and Monica S Lam, *Blocking and array contraction across arbitrarily nested loops using affine partitioning*, ACM SIGPLAN Notices **36** (2001), no. 7, 103–112.

[24] Preeti Ranjan Panda, Nikil D Dutt, and Alexandru Nicolau, *Efficient utilization of scratch-pad memory in embedded processor applications*, Proceedings of the 1997 European conference on Design and Test, IEEE Computer Society, 1997, p. 7.

[25] Louis-Noël Pouchet, Peng Zhang, P Sadayappan, and Jason Cong, *Polyhedral-based data reuse optimization for configurable computing*, Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, ACM, 2013, pp. 29–38.

[26] HPC PROJECT, *Par4all automatic parallelization*, 2012.

[27] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel, *Assigning program and data objects to scratchpad for energy reduction*, Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings, IEEE, 2002, pp. 409–415.

[28] Texas Instruments, *Multicore dsp+arm keystone ii system-on-chip (soc)*, 2 2012, Rev. D.

[29] Texas Instruments, *Throughput performance guide for c66x keystone devices*, 7 2012, Rev. A1.

[30] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor, *Polyhedral parallel code generation for cuda*, ACM Transactions on Architecture and Code Optimization (TACO) **9** (2013), no. 4, 54.

[31] Manish Verma, Stefan Steinke, and Peter Marwedel, *Data partitioning for maximal scratchpad usage*, Proceedings of the 2003 Asia and South Pacific Design Automation Conference, ACM, 2003, pp. 77–83.

[32] Manish Verma, Lars Wehmeyer, and Peter Marwedel, *Cache-aware scratchpad allocation algorithm*, Proceedings of the conference on Design, automation and test in Europe-Volume 2, IEEE Computer Society, 2004, p. 21264.

[33] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel, *Optimization of sparse matrix–vector multiplication on emerging multicore platforms*, Parallel Computing **35** (2009), no. 3, 178–194.

[34] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick, *Scientific computing kernels on the cell processor*, International Journal of Parallel Programming **35** (2007), no. 3, 263–298.

[35] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong, *Improving high level synthesis optimization opportunity through polyhedral transformations*, Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, ACM, 2013, pp. 9–18.

Appendix A

Synthetic Code to Build Model

```
1  float accu_f(void* restrict input, int cnt, int t) {
2      int i, j;
3      float *array = (float*) input;
4      float accu = 0;
5      _nassert((int) cnt % 8 == 0);
6      _nassert((int) array % 8 == 0);
7      for (j = 0; j < t; j++)
8          for (i = 0; i < cnt; i++) {
9              accu += array[i] * array[i] * array[i];
10         }
11
12     return accu;
13 }
```

Listing A.1: One Load per Iteration

```
1  float accu_f2(void* restrict input1, void* restrict
       input2, int cnt, int t) {
2      int i,j;
3      float *array1 = (float*) input1;
4      float *array2 = (float*) input2;
5      array1 = (float*)array1;
6      array2 = (float*)array2;
7      float accu = 0;
8      _nassert((int) cnt % 8 == 0);
9      _nassert((int) array1 % 8 == 0);
10     _nassert((int) array2 % 8 == 0);
11     for (j = 0; j < t; j++)
12         for (i = 0; i < cnt; i++) {
13             accu += array1[i] + array2[i];
14     }
15
16     return accu;
17 }
```

Listing A.2: Two Loads per Iteration

```
1  float accu_f3(void* restrict input1, void* restrict
       input2, void* restrict input3, int cnt, int t) {
2      int i,j;
3      float *array1 = (float*) input1;
4      float *array2 = (float*) input2;
5      float *array3 = (float*) input3;
6      array1 = (float*)array1;
7      array2 = (float*)array2;
8      array3 = (float*)array3;
9      float accu = 0;
10     _nassert((int) cnt % 8 == 0);
11     _nassert((int) array1 % 8 == 0);
12     _nassert((int) array2 % 8 == 0);
13     _nassert((int) array3 % 8 == 0);
14     for (j = 0; j < t; j++)
15         for (i = 0; i < cnt; i++) {
16             accu += array1[i] + array2[i] + array3[i];
17     }
18
19     return accu;
20 }
```

Listing A.3: Three Loads per Iteration

```
1  for (k = 1; k <= 100; k++) {
2
3      //EDMA load
4      if (n != 0)
5          for (j = 1; j <= n; j++) {
6              edma_trans(l2spm, ddr1, Sa, DMA_channel);
7              edmaWait4Completion(0);
8          }
9      //computation
10     for (j = 1; j <= m; j++)
11         fop(ddr2, Sb, 1);
12
13 }
```

Listing A.4: Mix of DMA transmission and Cached DDR Access

```
1  for (k = 1; k <= 100; k++) {
2
3      //EDMA load
4      if (b != 0)
5          for (j = 1; j <= b; j++) {
6              edma_trans(l2spm, ddr1, Sa, DMA_channel);
7              edmaWait4Completion(0);
8          }
9      //computation
10     for (j = 1; j <= a; j++)
11         fop(l1spm, Sb, 1);
12
13 }
```

Listing A.5: Code to Generate Regression Curve

Appendix B

Loop Tiled Kernels

```
1  for (t1 = 0; t1 <= floord(N - 1, 60); t1++) {
2      for (t2 = 0; t2 <= floord(N - 1, 60); t2++) {
3          for (t3 = 0; t3 <= floord(N - 1, 60); t3++) {
4              for (t4 = 60 * t1;
5                   t4 <= min(N - 1, 60 * t1 + 59);
6                   t4++) {
7                   for (t5 = 60 * t2;
8                        t5 <= min(N - 1, 60 * t2 + 59);
9                        t5++) {
10                       for (t6 = 60 * t3;
11                            t6 <= min(N - 1, 60 * t3 + 59);
12                            t6++) {
13                           C[t4][t5] = C[t4][t5] +
14                               A[t4][t6] * B[t6][t5];
15                       }
16                  }
17             }
18         }
19     }
20 }
```

Listing B.1: Matrix Multiplication Kernel

```
1  for (t1 = 0; t1 <= floord(N - 1, 60); t1++) {
2      for (t2 = 0; t2 <= floord(N - 1, 60); t2++) {
3          for (t3 = 0;
4               t3 <= floord(N - 1, 60); t3++) {
5              for (t4 = 60 * t1;
6                   t4 <= min(N - 1, 60 * t1 + 59); t4++) {
7                  for (t5 = 60 * t2;
8                       t5 <= min(N - 1, 60 * t2 + 59);
9                            t5++) {
10                     for (t6 = 60 * t3;
11                          t6 <= min(N - 1, 60 * t3 + 59);
12                               t6++) {
13                         C[t4][t5] += A[t6][t4] *
                               A[t6][t5];
14                     }
15                 }
16             }
17         }
18     }
19 }
```

Listing B.2: Ssyrk Kernel

```
1  for (t1 = 0; t1 <= floord(N - 2, 60); t1++) {
2      for (t2 = 0; t2 <= floord(N - 2, 60); t2++) {
3          for (t3 = max(1, 60 * t1);
4               t3 <= min(N - 2, 60 * t1 + 59); t3++) {
5               for (t4 = max(1, 60 * t2);
6                    t4 <= min(N - 2, 60 * t2 + 59); t4++) {
7                    C[t3][t4] =
8                         0.2 * (A[t3][t4] + A[t3][t4 - 1] +
9                              A[t3][1 + t4] + A[1 + t3][t4] +
10                                  A[t3 - 1][t4]);
11              }
12         }
13     }
14 }
```

Listing B.3: 2-D Jacobi Kernel

```
1  for (t1 = 0; t1 <= floord(N - 2, 60); t1++) {
2      for (t2 = 0; t2 <= floord(N - 2, 60); t2++) {
3          for (t3 = max(1, 60 * t1);
4               t3 <= min(N - 2, 60 * t1 + 59); t3++) {
5               for (t4 = max(1, 60 * t2);
6                    t4 <= min(N - 2, 60 * t2 + 59); t4++) {
7                    C[t3][t4] =
8                         0.2 * (A[t3][t4] + A[t3][t4 - 1] +
9                              A[t3][1 + t4] + A[1 + t3][t4] +
10                             A[t3 - 1][t4] + A[t3 - 1][t4 -
                                   1] +
11                             A[t3 + 1][1 + t4] +
12                             A[1 + t3][t4 - 1] +
13                             A[t3 - 1][1 + t4]);;
14              }
15          }
16     }
17 }
```

Listing B.4: 9 Point Stencil Kernel

```
1 for (t1 = 0; t1 <= floord(N - 2, 8192); t1++) {
2     for (t2 = max(1, 8192 * t1);
3          t2 <= min(N - 2, 8192 * t1 + 59); t2++) {
4          C[t2] = 0.2 * (A[t2] + A[t2 - 1] + A[1 + t2]);
5               }
6          }
7     }
8 }
```

Listing B.5: 1-D Jacobi Kernel