1-1-2010

Performance Analysis of Hybrid CPU/GPU Environments

Michael Shawn Smith
Portland State University

Performance Analysis of Hybrid CPU/GPU Environments

by

Michael Shawn Smith

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Karen Karavanic, Chair
Wu-chang Feng
Bryant York

Portland State University
2010

Abstract

We present two metrics to assist the performance analyst to gain a unified view of application performance in a hybrid environment: GPU Computation Percentage and GPU Load Balance. We analyze the metrics using a matrix multiplication benchmark suite and a real scientific application. We also extend an experiment management system to support GPU performance data and to calculate and store our GPU Computation Percentage and GPU Load Balance metrics.

Table of Contents

List of Tables

List of Figures

1. Introduction

Single core processors have hit a performance wall because of heat dissipation and power requirements. To overcome this the hardware industry started creating multicore CPUs. Even with multiple cores, these CPUs lag behind Graphic Processing Units (GPUs) in raw floating point performance. This is because the basic design philosophy between CPUs and GPUs is different. CPUs devote a large portion of available die space to control logic and cache memory. With GPUs a large portion of the die space is devoted to arithmetic units. Thus, GPUs are able to perform floating point operations faster.

The science community has taken notice of GPU performance, and has ported scientific codes to GPUs. Roeh et al. [35] have ported the two-point angular correlation function (TPACF) used in cosmological research achieving an approximate 80 times speedup. Phillips et al. [34] used GPUs to achieve a 7 times speedup for a molecular dynamic simulation. These applications have seen a many fold increase in performance. However, these efforts have been at small scale, utilizing one to two hundred compute devices. The high performance computing community has also taken notice of GPUs. Current multicore processors still limit the ability to achieve exascale computing because of power and heat dissipation. To move forward from the current petascale computing to exascale computing GPUs will be used because of their floating point performance and power consumption. This approach of adding GPUs as accelerators for CPUs is called *hybrid computing* [42].

Programming for GPUs is not easy because of the different execution model of GPUs. With GPUs the programmer must create a kernel that executes on the GPU, divide the data into blocks, and explicitly move the data and kernel to the GPU device. The kernel then executes asynchronously with code on the CPU. Most performance tools provide performance data on the GPU or the CPU in isolation. Therefore, the programmer has few tools available to talk about the overall performance of the program at a high level when working with multiple threads of execution and multiple kernel launches.

To provide a unified view of a program's performance we have developed two metrics targeting hybrid CPU/GPU environments. Our approach is novel because it takes into consideration both the CPU and GPU, and it works with currently available tools (in other words, we don't need an experimental device driver). This type of performance analysis reduces program development time, allowing programmers, domain scientists, to perform more real science.

Thesis Statement:

The GPU Load Balance and GPU Computation Percentage metric are a useful starting point for modeling performance of hybrid CPU/GPU systems: GPU Load Balance characterizes load balance between CPU and GPU or multiple GPUs; and GPU Computation Percentage indicates high overhead for data movement between the CPU and the GPU.

In this thesis we discuss our investigation of these two metrics, and our performance study using these metrics.

To calculate our metrics we gathered performance data using a GPU device centric performance tool and a performance tool that gathered performance data for the host CPU. We then combined this data and stored it in an experiment management system. The experiment management system was extended to support GPU performance data and to calculate and store values for the metric we were investigating.

The contributions of this work are:

1. implementation of a benchmark suite
2. a performance study in a hybrid environment
3. implementation of a GPU Load Balance metric
4. implementation of a GPU Computation Percentage metric
5. enhancement of an existing performance database tool to handle applications run in a hybrid environment

In the next section we discuss work related to ours.

2. Related Work

The most closely related work to ours includes benchmarks for hybrid environments and parallel performance tools.

2.1. Benchmarks

The Scalable Heterogeneous Computing (SHOC) benchmark suite [10], Rodinia [8], and the Parboil Benchmark suite [31] are a collection of benchmarks to test the performance of hybrid systems. Like our work these benchmarks report the performance of hybrid systems. However, these tools use benchmarks targeted for GPU performance problems, whereas we developed a benchmark suite for evaluating performance analysis methods and tools.

2.2. Parallel Performance Tools

Parallel performance tools related to our work can further be broken into CPU centric, GPU centric, and hybrid categories.

2.2.1. CPU Centric

There are a number of commercial and research parallel performance tools available. Commercial tools include Vampir [6]. The research community has developed a number of tools including HPC Toolkit [1], Scalasca [14], and TAU [38]. Of these, only TAU currently provides some support for gathering performance data in a hybrid environment.

2.2.2. GPU Centric

CudaProf [9] is Nvidia's performance tool used to generate profiles for kernel executions and memory transfers. It has a GUI front end and command line version.

To instrument an application, environment variables are set that indicate profiling is to be performance by the CUDA runtime system and the location of the configuration file. CudaProf has some limitations. The values reported for the performance counters do not represent the behavior of individual threads, but all the threads in a warp. Also, the profiler can only use one of the streaming multiprocessors on the device. Therefore, the performance counters don't even represent the behavior of all the warps. CudaProf can also impose significant overhead. If any profile counters are enabled all kernel calls are blocking. Normally kernel calls are non-blocking. This change in behavior could have a significant impact on performance if work on the host and the device are expected to be overlapped.

Nvidia Parallel Nsight [30] is a plugin for Visual Studio. It is currently in open beta and allows for profiling, tracing, and debugging of Nvidia devices.

In addition to measurement tools, work has been done in modeling. Hong and Kim [17] developed an analytical model for identifying performance bottlenecks. This model takes into consideration memory level parallelism. They developed two metrics for use with their model, memory warp parallelism and computation warp parallelism. Memory warp parallelism is a measure of the number of memory requests that can be handled concurrently. Computation warp parallelism is a measure of the work that can be completed while a warp is waiting for a memory request to complete. Another performance model for kernels running on GPUs was developed

by Baghsorkhi et al. [1] that used a work flow graph to estimate a kernel's performance. This work can be used with compilers to determine which optimizations to perform, and can allow a performance analysis to identify performance bottlenecks. Metrics introduced by this paper include SIMD pipeline latency, global memory latency, code divergence, and compute to global memory ratio.

Schaa and Kaeli [37] developed a method for predicting the performance of an application when multiple GPUs are used. A baseline is developed using a program that is deterministic and only uses a single GPU. Metrics were created for per element and per subset average execution time. An element is the smallest unit of computation, and a subset is an aggregation of elements. Predicted execution time for the multi-GPU version of the code is calculated using the per element average and the number of GPUs.

Ryoo et al. [36] have developed an auto-tuning approach that reduces the number of optimizations to consider when searching for an optimal configuration using the Pareto-optimal subset to prune the search space. They developed an efficiency and utilization metric that are used in determining the performance of a configuration. The efficiency of a configuration is a measure of the instructions that execute before the kernel finishes, and the utilization is a measure of compute resource usage. Measurements are done statically by analyzing PTX code and programmer supplied data.

Several other performance studies have been published. Matrix multiplication was used by Ryoo et al. [36] to do a performance study of various levels of optimization. This is similar to our work, but they utilize GFLOPS, global memory to computation cycles ratio, GPU execution percentage, CPU-GPU transfer percentage, kernel speedup on the GPU, and application speedup benchmarks in their study. We utilize our own metrics. They only performed their study on a single device where we use multiple device and overlap CPU and GPU in our study. Matrix multiplication on GPUs was studied by Allada et al. [1]. They compared the performance of matrix multiplication on a host CPU to the performance on a GPU. They also measured available bandwidth between host and device. An image registration application was ported by Bui and Brockman [7] to use GPUs. They did a performance study to measure the speedup of their implementation.

2.2.3. Hybrid

TAUcuda [27] is an extension to TAU [38] that allows TAU to profile and trace CUDA kernels running on Nvidia GPUs. This is an experimental approach because it relies on a call back interface that is, currently, only available in an experimental device driver from Nvidia. TAUcuda includes metrics for kernel execution elapsed time, memory transfer elapsed time, and memory transfer size. It also supports the default GPU counters.

Volkov and Demmel [43] did a performance study of matrix multiplication on GPU devices. There study also included using multiple devices and overlapping device and host computation. They also performed an in depth study of the device's memory system performance, startup time for kernels, and arithmetic throughput.
An equation to measure the amount of time to move data over the PCIe bus was also developed in this work.

Teodoro et al. [41] use a framework for decomposing an application into components that can be run on a GPU or CPU. A runtime system is used to determine if the component will be run on the host's CPU or the device. They studied application performance just using a GPU device, and in a hybrid environment using the CPU and GPU, and using a distributed memory system.

2.3. Conclusion

The novel aspects of our work are: our benchmark was implemented to test a performance tool instead of a GPU system; our work can be used in a hybrid environment with a production device driver; and our metrics can be applied to many applications in a hybrid environment.

3. Performance Metrics

Modeling the performance of hybrid systems is, currently, an open research question. Not a lot of work has gone into performance metrics for hybrid environments. In this thesis we attempt to lay down some foundation for reporting the performance of hybrid systems. To do this we define two metrics for performance of hybrid applications, GPU Computation Percentage and GPU Load Balance.

3.1. Background

Before discussing the metrics we'll give an overview of GPU programming. When writing applications targeted for GPUs the general design pattern is to allocate memory on the device, copy the data needed for the computation to the device, launch the kernel, and copy the results from the device to the host. Figure 1 shows a diagram of this design pattern.

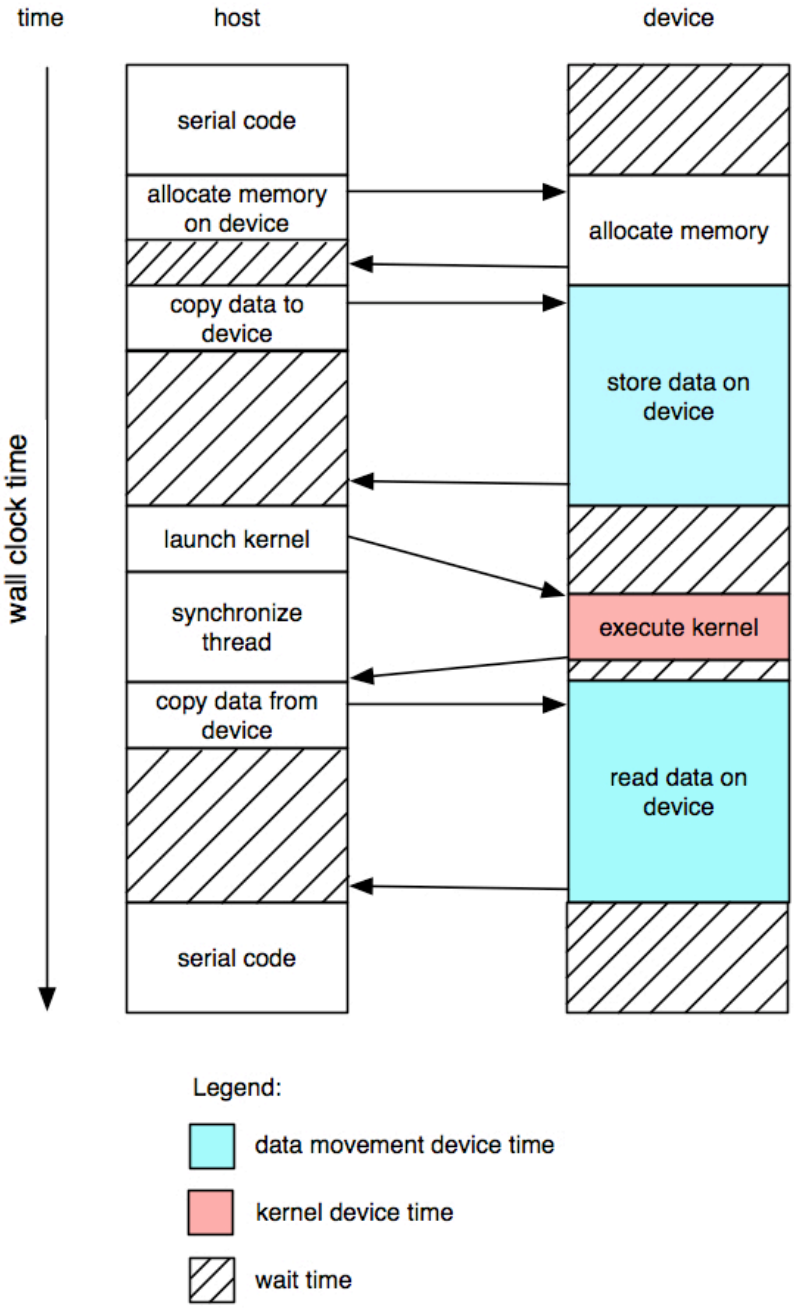


Figure 1: GPU Programming Design Pattern

Figure 2 is a sample program written in CUDA that uses this design pattern. Lines 1 through 3 define a kernel that just adds its arguments. Inside `main()`, lines 6 to 14 declare some variables. These lines would correspond to the "serial code" block in Figure 1. Memory is allocated on the device in lines 16 to 18 corresponding to the "allocate memory on dev" block in Figure 1. Data is copied to the device on lines 20 and 21 which corresponds to the "copy data to device" block in the figure. Lines 23 - 24 specify the number of threads that will execute the kernel. The launching of the kernel (line 25) corresponds to the "launch kernel" block in Figure 1. On line 27 we synchronize with the device so that the host will wait for the kernel to finish executing. This corresponds with the "synchronize thread" block in the figure. Finally the result is copied from the device to the host on line 29.

```
 1 __global__ void add(int *a, int *b, int *answer_dev) {
 2     *answer_dev = *a + *b;
 3 }
 4
 5 int main(int argc, char** argv) {
 6     int a_host;
 7     int *a_dev;
 8     int b_host;
 9     int *b_dev;
10     int answer;
11     int *answer_dev;
12
13     a_host = thread_id;
14     b_host = 3;
15
16     cudaMalloc((void **) &answer_dev, sizeof(int));
17     cudaMalloc((void **) &a_dev, sizeof(int));
18     cudaMalloc((void **) &b_dev, sizeof(int));
19
20     cudaMemcpy(a_dev, &a_host, sizeof(int),
cudaMemcpyHostToDevice);
21     cudaMemcpy(b_dev, &b_host, sizeof(int),
cudaMemcpyHostToDevice);
22
23     dim3 dimGrid(1);
24     dim3 dimBlock(1,32);
25     add<<<dimGrid, dimBlock, 0>>>(a_dev, b_dev, answer_dev);
26
27     cudaThreadSynchronize();
28
29     cudaMemcpy(&answer, answer_dev, sizeof(int),
cudaMemcpyDeviceToHost);
30
31     return 0;
32 }
```

Figure 2: Sample CUDA Application

3.2. GPU Computation Percentage Metric

Now that we have given a high level overview of GPU programming we'll cover our first metric, GPU Computation Percentage (GCP). Our goal for the GPU Computation Percentage metric is to indicate the amount of data movement overhead between the

CPU and GPU. For the GCP metric we wanted to use kernel wall clock time and data movement wall clock time. Kernel wall clock time would have been the amount of time the kernel executed on a device plus the amount of time spent moving the kernel to the device and any other overhead associated with a kernel launch. Data movement wall clock time would have been the wall clock time to move data to and from the device. However, kernel launches and data movement can occur asynchronously with the host so we are not able to measure these directly.

As an alternative we use kernel device time and data movement device time. Kernel device time is the amount of time a kernel executes on a device. This is reported as kernel `gputime` by CudaProf. Data movement device time is the amount of time data movement functions spend executing on the device. This is reported as function `gputime` by CudaProf for functions like `cudaMemcpy()`.

Data movement device time is not exactly the same as data movement wall clock time, but it does give some indication of how much wall clock time was spent moving data to and from the device. As seen in Table 1, empirical data shows that data movement time increases as the size of data increases. If we could directly measure data movement wall clock time it would also increase as the amount of data increases.

Table 1: Data movement device time

size (bytes)	host to device transfer time (µsec)	device to host transfer time (µsec)
1,024	5	5
1,048,576	342	318
4,194,304	1377	2041
16,777,216	5525	9545
67,108,864	22156	37200
268,435,456	88441	137025
1,073,741,824	353846	524745

The definition for the GPU Computation Percentage metric is shown in Equation 1. Kernel device time is the amount of execution time on the device for kernels, and device time is the amount of time kernel and data movement functions execute on the device.

$$\text{GCP} = \frac{\text{kernel device time}}{\text{device time}} \tag{1}$$

If no device time is spent moving data to the device the largest theoretical value of the metric is one. A value of one would indicate that no data was moved to the device, but this is unlikely because at a minimum some data needs to be moved from the host to the device for the device to do meaningful work. A high GPU Computation Percentage means that the data movement was effective. At the other extreme, if no kernel was launched on the device the value of the metric would be zero. A low GCP indicates that the developer needs to check data movement.

3.3. GPU Load Balance Metric

Our goal for the GPU Load Balance (GLB) metric is to show the load balance between the CPU and GPU or between multiple devices. To achieve this we use device time

and the application's wall clock time. Device time is the amount of time kernel and data movement functions execute on the device. Wall clock time is the amount of time the application ran. The definition of GPU Load Balance is shown in Equation 2.

$$GLB = \frac{\text{device time}}{\text{wall clock time}} \tag{2}$$

The range of values for this metric are zero to one. Values near zero indicate that a very small portion of the application executed on the device, and that the GPU device was mostly idle. A value of one indicates that the application used all of the GPU device's available execution time. This is a theoretical maximum because some time is consumed as overhead for the device's runtime environment on the host, i.e. allocating memory on the device and launching kernels. A high GPU Load Balance indicates a well balanced hybrid application or a device centric application. When multiple GPU devices are used the GLB also indicates the load balance between the devices.

This metric gives the programmer an indication of how much the application is using the device. This is helpful in a hybrid environment where the application performs work on the device and host. During development a value near zero would inform the developer that the application is hardly using the device, and prompt the developer that more investigation is needed to find parts of the code to run on the device to increase usage of the device. If no more of the application can be run on the device a value near zero would indicate that it may be faster to run the application strictly on

the host. This could happen because overhead to run the kernel on the device is exceeding the amount of time the kernel executes on the device. A value near one indicates that the application is using the device for the majority of the execution time.

When used in a system with multiple devices a metric value is reported for each device. In the multiple device environment the range of values is the same as discussed above, but is specific for each device. This gives the programmer the added benefit of seeing the distribution of work among the devices. For example, if one device has a high GPU Load Balance and the other has a GLB of zero, the second device is not being used at all and the programmer may want to investigate a better division of work to improve performance.

A limitation of this metric is that it does not indicate how well optimized an application is for the device or host. For example, a poorly tuned application could consume a large portion of device time creating a GPU Load Balance near one. However, if the application was optimized for the device, the GLB would decrease. The metric does not distinguish these two cases.

4. Benchmark Suite

To investigate the utility of our metrics we wanted to use a benchmark. We needed an easily understandable benchmark that could have various levels of optimizations applied to it and studied. We also wanted to be able to easily verify the correctness of the benchmark. Our starting point was a single device implementation of matrix multiplication described by Kirk and Hwu [22]. We took the single device implementation and extended it to support multiple GPU devices, overlap CPU and GPU computation, use pinned memory, and use asynchronous memory transfers. We feel this work may be of benefit to others studying GPU performance tools because it gives them an easily understandable benchmark that includes common GPU optimization strategies. (Note: since the start of our project other benchmarks have been published [8, 10, 31].)

4.1. Single Device Matrix Multiplication

In this section we describe how matrix multiplication is performed using a single GPU device and the optimizations applied to matrix multiplication as described by Kirk and Hwu [22]. We use a 4 x 4 matrices to illustrate the process. M and N are the two matrices to be multiplied and the result is stored in P.

First, memory must be allocated and initialized on the host. The matrices M, N, and P are allocated using malloc() as a one-dimensional array. Since the matrices being multiplied are two-dimensional and the memory used to store the matrices is one-

dimensional, a mapping is needed to map between an element's location in the matrix and its location in memory. Equation 3 shows the formula to calculate an element's index in memory allocated on the host. Row is the row the element is in, column is the column the element is in, and width is the width of the matrices.

$$\text{memory index} = \text{row} \times \text{width} + \text{column} \tag{3}$$

To find its row number Equation 4 is used:

$$\text{matrix row} = \left\lfloor \frac{\text{index}}{\text{width}} \right\rfloor \tag{4}$$

To find its column number Equation 5 is used:

$$\text{matrix column} = \text{index} \% \text{width} \tag{5}$$

After the memory is allocated M and N are initialized to a default value. Both M and N are initialized using Equation 6. Matrix P is left uninitialized because the result generated by the device is copied from the device into this matrix.

$$\text{matrix}[\text{row}][\text{column}] = \text{row} + \text{column} + 1 \tag{6}$$

Next, memory must be allocated on the device to store the M and N matrices, and memory needs to allocated on the device to store the result. Memory is allocated on the device using cudaMalloc() of the same size as allocated on the host. The matrices on the device are named M_d, N_d, and P_d. Now that memory is allocated on the device the M and N matrices are copied from the host to the device. The matrix multiplication kernel (discussed below) is launched, the result calculated, and the result is stored in P_d on the device. The P_d matrix is then copied from the device into

matrix P on the host. Once the result is in P the matrix can be printed or stored to disk.

Three different levels of optimizations are applied to the matrix multiplication kernel. The optimization levels are naive, tiled, and tiled+shared memory.

4.1.1. Naive Kernel

The naive implementation described by Kirk and Hwu [22] does a straight forward dot product calculation. This kernel is configured with a grid size of one block. Therefore, the size of matrix this kernel can calculate is limited by the maximum number of threads that can be in a block. In the GPU devices used in this thesis, the maximum number of threads in a block is 512 threads. Thus, the largest matrix that this kernel can calculate, that is a power of two, is 16 x 16. To overcome this limitation a tiled kernel was implemented, as discussed in the next section.

4.1.2. Tiled Kernel

To overcome the limited size of matrices that can be used with the naive implementation a tiled kernel was implemented. The tiled implementation, as described by Kirk and Hwu [22], breaks the P_d matrix up into tiles that are the same dimension as the block size. Each thread block then works on a smaller portion of the matrices. Figure 3 shows how the 4 x 4 P_d matrix is divided into four blocks of size 2 x 2.

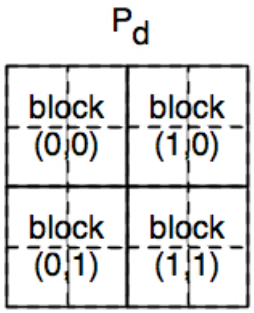


Figure 3: P_d tiled

Figure 4 demonstrates how element (0,0) in P_d is calculated. The (0,0) element in P_d is calculated by thread (0,0) in block (0,0) by doing a dot product calculation of the first row in M_d and the first column in N_d. Since each thread block is working on a small portion of P_d a larger matrix multiplication may be done.

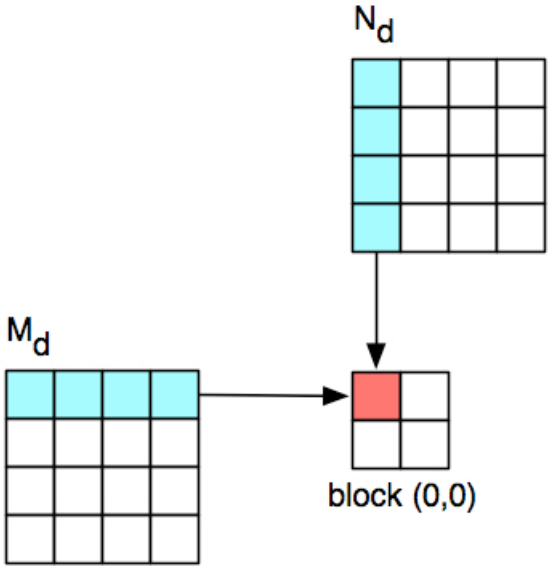


Figure 4: Calculation of element (0,0) in block (0,0)

4.1.3. Tiled + Shared Memory Kernel

The tiled kernel allows larger matrices to be multiplied, but it accesses data in global memory multiple times. When the M and N matrices are copied into M_d and N_d on the

device they reside in global memory. Global memory is larger, but slower than shared memory on the device. As demonstrated in Figure 5 the tiled kernel access the same data in M_d and N_d multiple times. The element marked with a 2 in block (0,0) access the same row in M_d as the element marked with a 1. This means that to calculate element 1 and 2 the first row of M_d is accessed twice. If this row could be stored in shared memory the number of accesses to global memory would be decreased, improving performance. Unfortunately, shared memory is a scarce resource and for larger matrices cannot fit all of the rows from M_d and columns from N_d needed to perform the dot product.

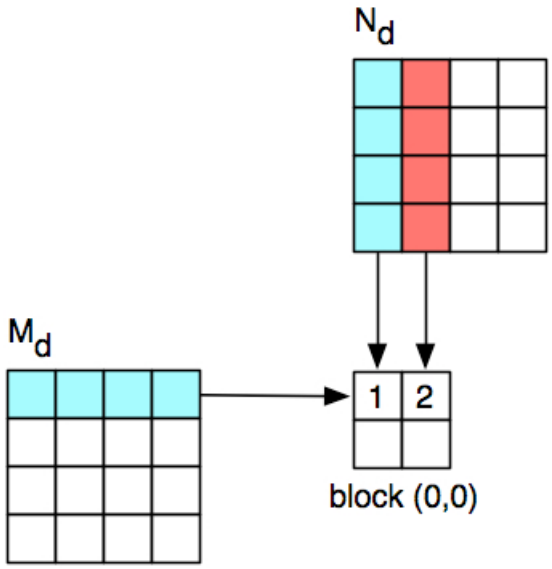


Figure 5: Tiled kernel memory access pattern

To overcome the problem of limited shared memory, the M_d and N_d matrices are tiled and the result is calculated in phases as described by Kirk and Hwu [22]. The tiles of M_d and M_d are moved into shared memory named M_{ds} and M_{ds} respectively. The first

phase of our 4 x 4 matrix multiplication is shown in Figure 6. In this phase a 2x2 tile of M_d is moved into M_{ds} and a 2x2 tile of N_d to moved into N_{ds} and a dot product is performed with the values in M_{ds} and N_{ds}. The partial result is stored in P_{value1}.

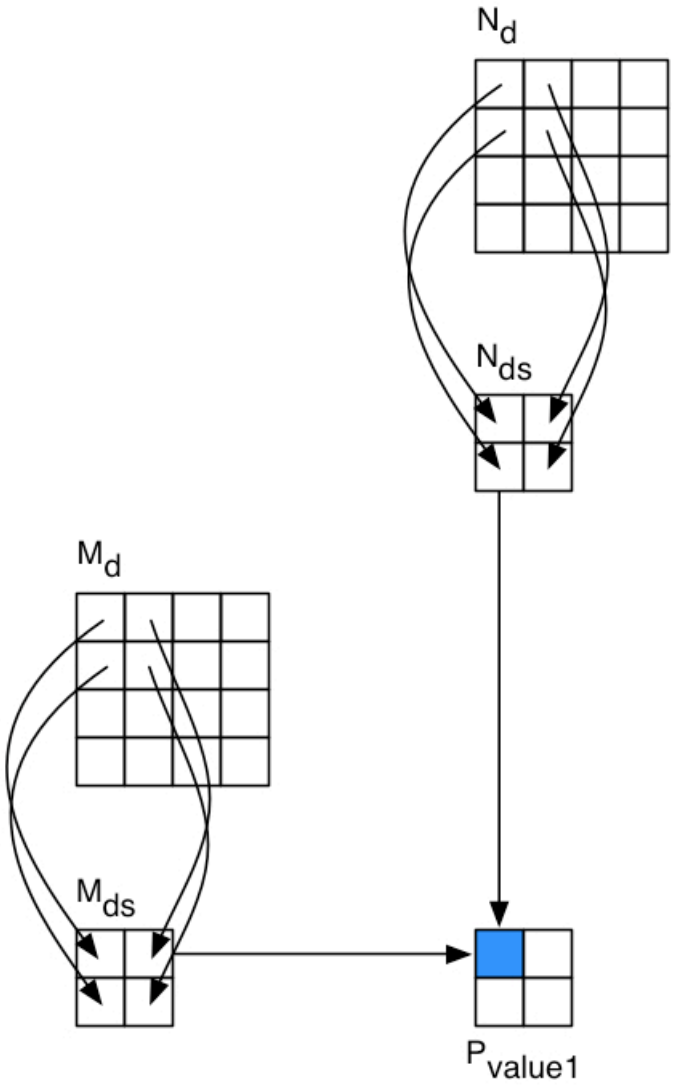


Figure 6: Tiled+shared kernel phase 1

The second phase of the matrix multiplication is shown in Figure 7. In this phase another 2x2 tile of M_d is moved into M_{ds} and another 2x2 tile of N_d is moved into N_{ds}

and a dot product is performed with the values in M_{ds} and N_{ds}. The partial result is stored in P_{value2}. The final result is stored in P_d by adding P_{value1} and P_{value2}.

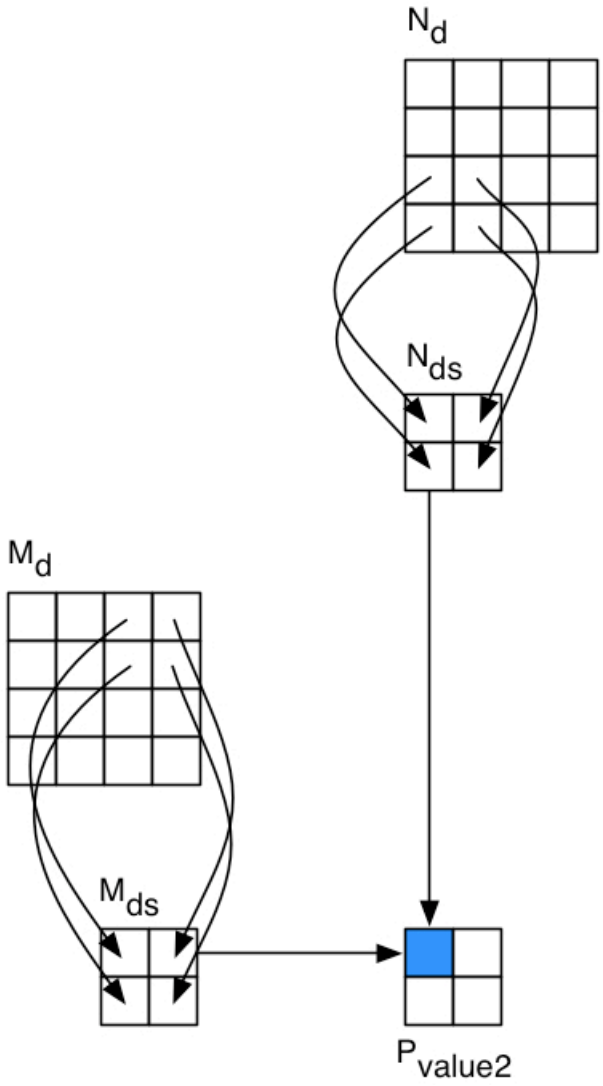


Figure 7: Tiled+shared kernel phase 2

4.2. Multiple Device Matrix Multiplication

In this section we describe how we extended the single device matrix multiplication described in Section 4.1. The CUDA programming model requires at least one host threads per device. This means that each host thread needs to specify which device its

kernels are to be executed on. The default is device 0. Multiple host threads may specify the same device, but a host thread can only use one device at a time. Therefore, for a single application to use multiple devices it must create multiple threads of execution using OpenMP or pthreads. In this thesis we use pthreads. Figure 8 shows how the calculation of P is divided between two threads. The blue elements in Figure 8 are allocated to thread 0 and the red elements are allocated to thread 1. Thread 0 calculates the top half of P and thread 1 calculates the bottom half of P. Therefore thread 0 needs the top half of M and all of N, and thread 1 needs the bottom half of P and all of N.

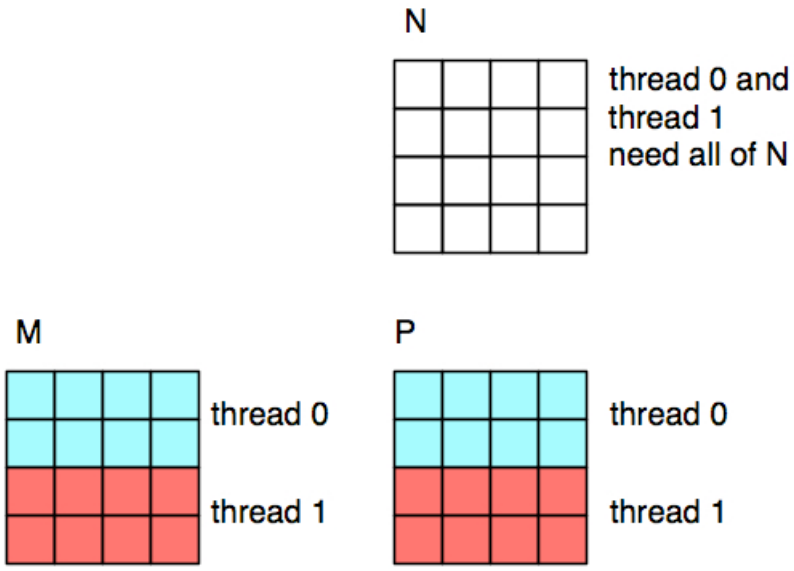


Figure 8: Division of work between two threads

After the work has been divided each thread calculates its portion of P as shown in Figure 9. Each thread copies its portion of M and all of N to M_d and N_d on its device. Each thread launches the matrix multiplication kernel, and the thread's partial result is

stored in P_d. The matrix multiplication kernels are the same kernels outlined in Sections 4.1.1, 4.1.2, and 4.1.3. Next, the thread's partial result is moved from the device into the portion of P, on the host, that it calculated. Blue elements in P are calculated by thread 0 and red elements are calculated by thread 1. The result in P can then be stored on disk or checked for accuracy.

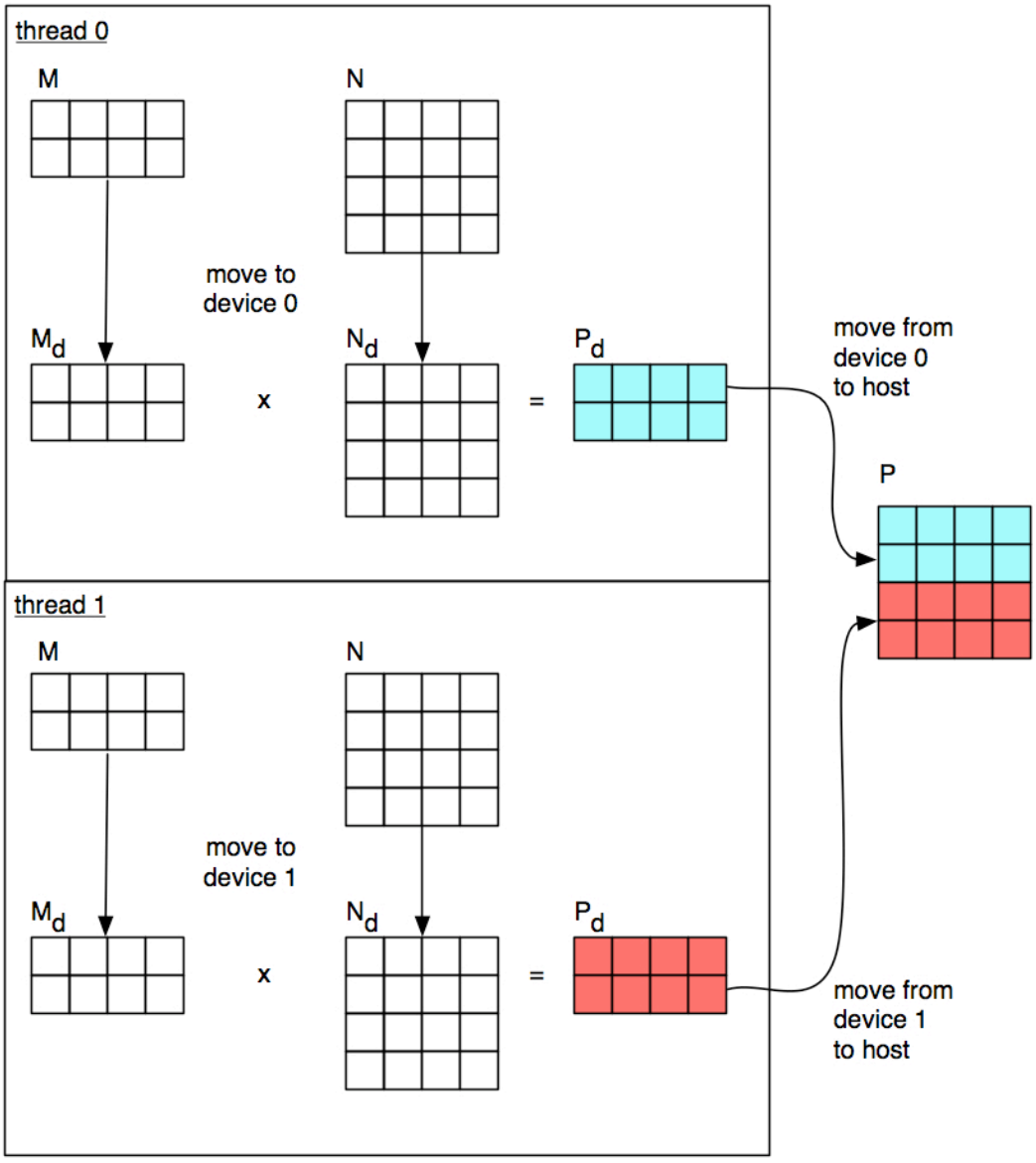


Figure 9: Multi-device matrix multiplication

4.3. Memory Optimizations

We extended the matrix multiplication benchmark to use pinned host memory, also called page-locked host memory. This memory optimization allocates memory that won't be paged by the operating system. If the system has a front side bus this increases the bandwidth between the host and device [29].

We also extended the matrix multiplication to use asynchronous memory transfers. Asynchronous memory transfers are non-blocking on the host allowing the host to perform other work while the data is being transferred to and from the device. Pinned memory is required by CUDA for asynchronous memory transfers.

4.4. Hybrid Matrix Multiplication

We extended the matrix multiplication benchmark to work in a hybrid environment. We modified the code so that part of the multiplication is performed on the device and part on the host processor. We accomplish this by partitioning the matrices as described in Section 4.2. However, instead of having two devices perform the calculation, one device performs part of the calculation and the host performs the other part. The partial results are then combined as usual.

4.5. Problems Implementing Matrix Multiplication

In this section we discuss some of the challenges we encountered implementing matrix multiplication for GPUs. The initial version of matrix multiplication created a file with the matrices to be multiplied and a gold file. The gold file contained a known

correct result. The gold file was calculated using a simple matrix multiplication function on the host. The matrix multiplication was performed using matrices with elements of a float type. When the results where written to file they were cast to an unsigned int, and when the result was read from the file they were cast back to a float.

The naive, tiled, and tiled plus shared memory implementations with this design worked with matrices up to 4096x4096. They all produced results calculated on a GPU device that matched the gold result. However, when we tried to extend the initial version to use multiple devices, pinned memory, or asynchronous memory transfers it would produce incorrect results for matrix sizes above 1024x1024. Inspecting the incorrect result it was seen that part of the result was correct. We suspected the cause of the bug was the type casting used during reading and writing of the matrices from files. To test this theory we reimplemented the matrix multiplication using strictly unsigned int types and removing all the type casting. This change immediately produced correct results for all optimization of matrix multiplication used and on all sizes of matrices used. Our suspicion is that the type casting introduced rounding errors that become larger as the matrix multiplication was performed.

5. PerfTrack

We needed a method to calculate, store, and analyze our GPU Computation Percentage and GPU Load Balance metrics. PerfTrack [21] was chosen to manage the results for our metrics. PerfTrack is an experiment management system designed for importing, storing, retrieving, and analyzing machine and performance data. For example, it can be used to compare the GPU Computation Percentage of the matrix multiplication benchmark across different versions of the CUDA runtime system. However, it needed to be extended to support hybrid system data and GPU performance data.

5.1. Work Done Extending PerfTrack

In this section we discuss work done to extend PerfTrack to import machine and GPU performance data.

5.1.1. Machine Data

PerfTrack's machine data gathering scripts were extended to search for GPU devices on a node. To support GPU performance data the machine resource hierarchy, which describes a system, was updated to include GPU devices. Next, a utility was implemented that queried a device for its properties. The machine data gathering scripts were then updated to use this query utility to add resource information for the GPU devices on the node. Figure 10 shows the resource information automatically gathered by our extension to the machine data gathering scripts.

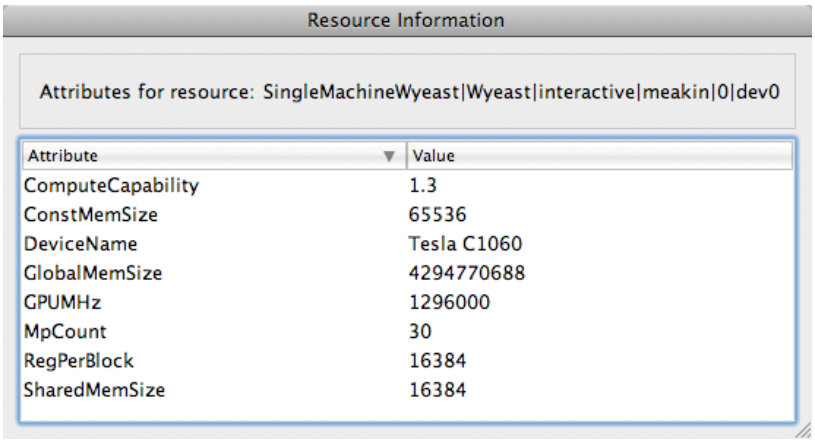


Figure 10: GPU Device Resource Information

5.1.2. Build Data

Before PTdFgen.py can be used to create a PTDF for performance results, a build file and run file are needed. These files were created manually as outlined in PerfTrack's documentation.

5.1.3. Performance Results

To import a CUDA Prof log into PerfTrack we need to be able to convert a CUDA Prof log entry into a PerfTrack performance result. Each CUDA Prof log entry contains a time stamp and method name, followed by the metric values gathered. A log entry is shown in Figure 11. This entry shows the time the MatrixMulKernel2 kernel was launched, the GPU time, CPU time, grid size, block size, and static shared memory per block metrics.

```
timestamp=[ 610.000 ] method=[ _Z16MatrixMulKernel2PfS_S_i ]
gputime=[ 17.120 ] cputime=[ 40.000 ] gridSize=[ 1, 1 ]
blockSize=[ 16, 16, 1 ] staSmemPerBlock=[ 44 ]
```

Figure 11: CUDA Prof log entry

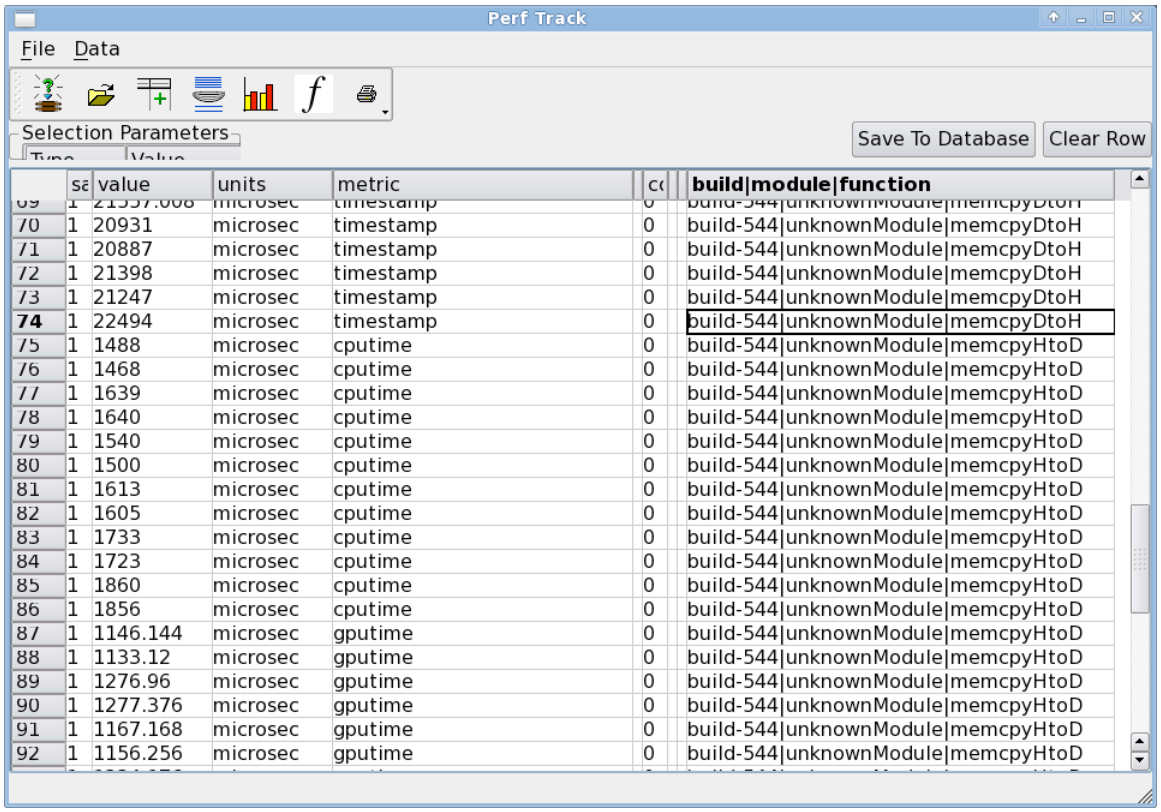
To convert the CUDA Prof log entry we need to map the log entry to a PerfTrack performance result. A PerfTrack performance result consists of a context, performance tool, metric, metric value, units, start time, and stop time. A context is created for each method in the CUDA Prof log and the method in the log entry is associated with this context. A performance result is created for each metric in the CUDA Prof log entry. The performance tool for each performance result is "CudaProf." The metric value is the value in the log entry between the "[" and "[". The units for the performance entry are based on the metric. For example, time metrics are in microseconds and memory size metrics are in bytes. The start and stop attributes of a performance result are not used and left as "noValue."

The basic strategy to parse the CUDA Prof log file is to find each log entry and to extract the metric and metric values from the entry. To extract the metric and its value, a list of regular expressions for the known metrics is iterated over. Each metric regular expression in the list is used to search the log entry. If a match is found, a performance result is created for the found metric and the performance result is added to the execution.

This procedure allows PTdFgen.py to generate a PTDF file. However, as initially implemented the PTDF would not successfully load into PerfTrack. Some metric values in the CUDA Prof log are multidimensional. For example, the value for the block size metric is a two dimensional result of the height and width of the kernel's

block configuration. To overcome this problem, the single block size metric was broken into two metrics, a block size X metric and a block size Y metric. Another problem encountered with the initial implementation was with the gpu start timestamp and gpu end timestamp metrics. As reported by CUDA Prof, these metrics have a hex value. For example, 11b46a729843c880. As the performance results in the PTDF file are loaded into PerfTrack they are cast to a float. This creates a problem because these hex values can't be immediately cast to floats. To solve this problem the hex values were first converted to an integer which in turn could be cast to a float as the PTDF was loaded into PerfTrack.

Our initial approach causes us to lose information: there is no way to associate a specific performance result with one particular function call. Figure 12 shows the performance results gathered for a run of the matrix multiplication application. Several performance results are shown for the CPU time metric for the memcpyHtoD method, but there is no way to see which call to memcpyHtoD the metric value is associated with. In the future we plan to tag the log data as we parse it.



	sa	value	units	metric	cc	build|module|function
70	1	20931	microsec	timestamp	0	build-544|unknownModule|memcpyDtoH
71	1	20887	microsec	timestamp	0	build-544|unknownModule|memcpyDtoH
72	1	21398	microsec	timestamp	0	build-544|unknownModule|memcpyDtoH
73	1	21247	microsec	timestamp	0	build-544|unknownModule|memcpyDtoH
74	1	22494	microsec	timestamp	0	build-544|unknownModule|memcpyDtoH
75	1	1488	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
76	1	1468	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
77	1	1639	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
78	1	1640	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
79	1	1540	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
80	1	1500	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
81	1	1613	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
82	1	1605	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
83	1	1733	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
84	1	1723	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
85	1	1860	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
86	1	1856	microsec	cputime	0	build-544|unknownModule|memcpyHtoD
87	1	1146.144	microsec	gputime	0	build-544|unknownModule|memcpyHtoD
88	1	1133.12	microsec	gputime	0	build-544|unknownModule|memcpyHtoD
89	1	1276.96	microsec	gputime	0	build-544|unknownModule|memcpyHtoD
90	1	1277.376	microsec	gputime	0	build-544|unknownModule|memcpyHtoD
91	1	1167.168	microsec	gputime	0	build-544|unknownModule|memcpyHtoD
92	1	1156.256	microsec	gputime	0	build-544|unknownModule|memcpyHtoD

Figure 12: PerfTrack Performance Results

5.1.4. Wall Clock Time

A CUDA Prof log file does not include the application's wall clock time for the application's entire execution. To overcome this we used the time utility to measure the application's wall clock time. The output from time was appended to the CUDA Prof log file.

During parsing of the log file, as explained above, the output from the time utility was searched for. If it was found, the wall clock time was extracted from the output, and a performance result added to the execution resource in PerfTrack. This solution

worked for applications that used a single device, but when multiple devices were used a problem became evident.

When multiple devices are used, CUDA Prof needs to be configured to output the log data for each device into a separate log file. Thus, if an application uses two devices there should be two log files, with each log file containing the performance data for a GPU device. We changed our data loading step to load wall clock time independently instead of appending to the log file.

5.1.5. Multiple CUDA Prof Log Files

Another problem was discovered after CUDA Prof was configured to generate a log file for each device. When multiple devices were used an additional log file was created. For example, if two devices were used three log files were created. The first log file would contain header information specifying it was for device 0, but contain no performance data. The second log file would contain header information specifying it was for device 0 and contain the performance data for device 0. The third file would contain the header information and performance data for the second device. We assume the header correctly identifies the device per log and ignore the log file with no data.

5.1.6. Metric Calculation

In this section we discuss how PerfTrack was extended to calculate the GPU Computation Percentage and GPU Load Balance metrics. The GPU Computation Percentage metric is discussed in Section 7, and the GPU Load Balance is discussed in

Section 8. As described in Section 5.1.3 the CUDA Prof log files are parsed to extract performance data gathered by CUDA Prof and import it into PerfTrack. During this procedure the application's wall clock time is also imported into PerfTrack. Since the data needed to calculate the metric is available at this time it was decided to calculate the value of the metrics. Doing the calculation at this time instead of when data is retrieved from PerfTrack provides a performance benefit because the metric's value isn't recalculated each time the user requests this information. It can be immediately retrieved from the database.

The toolParser.py module was extended to calculate the metric values. A method was added that searches the log file for the needed information to calculate the metric values. Once found it calculates the metric values and adds them to the execution resource. By adding it to the execution resource it makes the metric available in PerfTrack when performance data on the execution is retrieved.

5.1.7. Conclusion

In this section we have discussed the work we've done so far to import CUDA Prof log files into PerfTrack. Several problems were found:

- multidimensional metric values
- nonnumeric metric values
- associating performance result with specific function call
- extraneous CUDA Prof log file

The solution to the multidimensional metric values and metric hex values have been implemented. The problem with associating performance results with a specific function call in the call graph is left for future work.

6. Experimental Design

In this section we discuss our experiment goals and how we designed our experiments. The goals of our experiments are to investigate the utility of the GPU Computation Percentage and GPU Load Balance metrics. We do this by performing a performance study of a simple matrix multiplication benchmark and a real scientific application in a hybrid environment. We then analyze the results from the experiments.

In this section we discuss:

- the sizes and configurations of matrices used
- background information on the scientific application used
- how the scientific application was configured
- how the matrix multiplication benchmark and scientific application where instrumented
- the system the experiments were run on

We begin by discussing how we implemented the matrix multiplication benchmark.

6.1. Matrix Multiplication Benchmark

In this section we cover how the matrix multiplication benchmark was configured for our experiments. Table 2 shows the dimensions and memory sizes of matrices used. A 16 x 16 matrix was chosen because it is the largest matrix supported by the naive matrix multiplication kernel as discussed in Section 4.1.1. The smallest matrix size supported by the tiled and tiled plus shared memory kernels is 16 x 16. This is

because the matrix needs to be at least as large as the tile used in these optimizations, which is 16 x 16. A 32 x 32 matrix is included so that each thread used, in the multiple device portion of this case study, has at least the minimum sized matrix needed for the tiled and tiled plus share memory optimized kernels. All elements in the matrices are unsigned int data types.

Table 2: matrix sizes

Dimension	memory size (bytes)
16 x 16	1,024
32 x 32	4,096
512 x 512	1,048,576
1,024 x 1,024	4,194,304
2,048 x 2,048	16,777,216
4,096 x 4,096	67,108,864
8,192 x 8,192	268,435,456
16,384 x 16,384	1,073,741,824

Several levels of optimization are investigated. We use a naive, tiled and tiled plus share memory optimized kernels, and three levels of memory optimizations are used with each kernel: paged, pinned, and pinned with asynchronous memory transfers. Each kernel optimization/memory optimization combination was executed 105 times. These runs were performed on one and two GPU devices.

6.2. NAMD

In our experiments we used a full scientific application, called NAMD [33]. NAMD is an application for molecular dynamic simulations.[1] It was extended by Phillips et

[1] NAMD was developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign.

al. [34] for use with GPUs. We chose NAMD because it is a real world application for doing real science that has already been ported to use GPUs. We configured NAMD to simulate the Satellite Tobacco Mosaic Virus (STMV) [13]. We downloaded the example STMV simulation from http://www.ks.uiuc.edu/Research/namd/utilities/stmv/. The simulation configuration was modified by changing the following configuration variables to the specified values:

- timestep 0.15
- fullElectFrequency 5
- stepspercycle 5
- outputEnergies 100
- numsteps 120

NAMD's support for GPUs is still under development. So, these configuration values were obtained through trial and error so that we could complete a full simulation.

6.3. Instrumentation

CudaProf was used to instrument the application, and gather device performance data. The profiler was configured to gather timestamps. By selecting just timestamps it automatically includes the `gputime` and `cputime` for each method executed on the device and for kernel calls. We ran the application on wyeast01 and wyeast02 as described in Section 6.4.

To gather wall clock time we used the external `time` [40] command. It has a resolution of a hundredth of a second. The `gettimeofday()` system call with a resolution of a microsecond was considered after the results were gathered. It was not used because it would have required modification of the scientific application's source code to insert the system call and output the timing results. The application's wall clock time could have also been calculated using CUDA event streams. Event streams have a resolution of milliseconds. The disadvantage to using CUDA event streams is that it would have made our methodology CUDA specific and would have required modification to the scientific application's source code as well.

6.4. Wyeast

We used a portion of Wyeast in our experiments. Wyeast is a cluster in the High Performance Computing Lab at Portland State University. Our experiment system consisted of two nodes from Wyeast named wyeast01 and wyeast02, a node named meakin was used as a login and testing node, an Nvidia S1070, and a Netgear gigabit switch used to connect the three nodes. The Nvidia S1070 consists of four T10 compute devices. Two devices are connected to wyeast01 and wyeast02 via a host bus adapter. The configuration for meakin is described in Table 3.

Table 3: meakin configuration

CPU	(2) Intel Xeon E5504 2.0GHz
Memory	12GB
OS	Ubuntu server 9.04 (Jaunty Jackalop)
Linux Kernel	2.6.28.18
Nvidia Driver	195.36
CUDA Runtime Version	3.0
Device 0	Tesla C1060
	CUDA Capability: 1.3
	Memory: 4GB
Device 1	Quadro NVS 295
	CUDA Capability: 1.1
	Memory: 256MB

The configuration for wyeast01 and wyeast02 is described in Table 4.

Table 4: wyeast01 and wyeast02 configuration

CPU	Intel Xeon E5520 2.27GHz
Memory	12GB
OS	CentOS 5.4
Linux Kernel	2.6.18.164
Nvidia Driver	195.36
CUDA Runtime Version	3.0
Devices	(2) Tesla T10 Processor

The configuration for the S1070 is shown in Table 5.

Table 5: Tesla S1070 configuration

Devices	(4) Tesla T10 Processor
	CUDA Capability: 1.3
	Memory: 4GB

7. GPU Computation Percentage Case Study

Our target environment is a hybrid environment that consists of host CPUs and GPU devices used as accelerators. The host could have multiple GPU devices available, or the environment could be a cluster with devices attached to each host node.

We wanted a tool that would assist the developer while writing programs for this hybrid environment. One performance bottleneck for hybrid environments is moving data between the host and device. We wanted a tool that would tell the developer if the additional computation completed on the device was worth the cost in overhead.

To achieve this goal we considered several metrics including acceleration execution time, unaccelerated execution time, speedup, kernel executions per second, and percentage of PCIe bus bandwidth. Initially we wanted to measure when data was being moved between the host and device, and when data was being moved between devices. We felt that being able to measure this, it would help use develop the tool needed to assist the developer writing programs for a hybrid environment.

We investigated for a means to measure the amount of data moving across the PCIe bus and couldn't find an acceptable solution. CudaProf reports the amount of data moved to and from the device for memory copies, but it doesn't report data movement for non-memory copies, e.g. kernel calls and device management functions. Also memory copies can occur asynchronously which would make it problematic to

correlate when data was being transferred to/from multiple devices. Another idea was to instrument the CUDA runtime itself; however, the CUDA runtime is essentially a black box because it is close source proprietary software. We also considered hardware to instrument the PCIe bus, but we did not have this in our budget and we wanted to develop a software based solution.

We eventually settled on two metrics that could be calculated using results form CudaProf or any other tool that could instrument a GPU; GPU Computation Percentage and GPU Load Balance. Our goal for the GPU Computation Percentage metric is to indicate the amount of data movement overhead between the CPU and GPU. The definition of GPU Computation Percentage (1) is repeated below.

$$GCP = \frac{\text{kernel device time}}{\text{device time}} \quad (1)$$

7.1. Single Device Results

Table 6 to Table 8 show the mean GPU Computation Percentage for the matrix multiplication benchmark using a single device. For the data in Table 6 no memory optimization was used, in Table 7 pinned memory was used, and in Table 8 pinned memory with asynchronous memory transfers was used. The naive column shows the metric value for naive kernel, the tiled column shows the metric values for the tiled kernel, and the tiled+shared column shows the metric values for the tiled with shared memory kernel.

First we examine the naive kernel. The mean GPU Computation Percentage for all three memory optimizations was very similar, ranging from 0.52 to 0.56.

Next we examine the tiled kernel. The mean GCP for the 16x16 problem size ranged from 0.52 to 0.57. At this problem size it may be faster to do the computation strictly on the host. For problem sizes 512x512 and larger the mean GCP ranged from 0.95 to 1.00. In all of these cases, the device spent more time doing work than was spent transferring data; therefore, the overhead of transferring data to the device was justified.

Examining the tiled plus shared memory kernel, the mean GPU Computation Percentage for the 16x16 problem size ranged from 0.40 to 0.45. For problem sizes 512x512 and larger the mean GPU Computation Percentage ranged from 0.75 to 0.99. Comparing the GPU Computation Percentage of the tiled kernel and the tiled plus shared memory kernel, the GPU Computation Percentage was lower for the tiled plus shared memory kernel in all cases. When an optimization is applied to a kernel the kernel's device time will decrease, but the data movement device time will stay the same. This causes the GPU Computation Percentage to decrease even though there was a speedup in the benchmark.

Comparing the GPU Computation Percentage for the matrix multiplication benchmark using paged memory, pinned memory, and pinned memory with asynchronous memory transfers we did not see a noticeable difference. This is because of how the benchmark was implemented. Asynchronous memory transfers are nonblocking for the host. This allows the host to continue working while data is transferred to the device. Even with the asynchronous memory transfers the device still needs all the data before it can continue and the host still waits for work on the device to complete before it continues.

Table 6: GPU Computation Percentage, paged memory single device

matrix size	naive	tiled	tiled+shared
16x16	0.56	0.57	0.45
512x512	-	0.95	0.75
1024x1024	-	0.96	0.82
2048x2048	-	0.97	0.88
4096x4096	-	0.99	0.94
8192x8192	-	0.99	0.97
16384x16384	-	1.00	0.99

Table 7: GPU Computation Percentage, pinned memory single device

matrix size	naive	tiled	tiled+shared
16x16	0.52	0.52	0.40
512x512	-	0.95	0.75
1024x1024	-	0.97	0.84
2048x2048	-	0.98	0.91
4096x4096	-	0.99	0.95
8192x8192	-	0.99	0.98
16384x16384	-	1.00	0.99

Table 8: GPU Computation Percentage, pinned memory, asynchronous memory transfers, single device

matrix size	naive	tiled	tiled+shared
16x16	0.52	0.52	0.40
512x512	-	0.95	0.75
1024x1024	-	0.97	0.84
2048x2048	-	0.98	0.91
4096x4096	-	0.99	0.95
8192x8192	-	0.99	0.98
16384x16384	-	1.00	0.99

7.2. Multiple Device Results

Table 9 to Table 11 show the mean GPU Computation Percentage for the matrix multiplication benchmark using two devices. For the data in Table 9 no memory optimization was used, in Table 10 pinned memory was used, and in Table 11 pinned memory with asynchronous memory transfers was used. The naive column shows the metric value for naive kernel, the tiled column shows the metric values for the tiled kernel, and the tiled+shared column shows the metric values for the tiled with shared memory kernel.

First we examine the naive kernel. The mean GPU Computation Percentage for all three memory optimizations was very similar. Ranging from 0.55 to 0.65. At this problem size it may be faster to run the computation strictly on the host.

Next we examine the tiled kernel. The mean GPU Computation Percentage for the 32x32 problem size ranged from 0.60 to 0.64. Like the naive kernel, it may be faster to do the computation strictly on the host for this problem size. For problem sizes

512x512 and larger the mean GPU Computation Percentage ranged from 0.93 to 1.00. In all of these cases, the device spent more time doing work than was spent transferring data; therefore, the overhead of transferring data to the device was justified.

Examining the tiled plus shared memory kernel, the mean GPU Computation Percentage for the 16x16 problem size ranged from 0.43 to 0.47. For problem sizes 512x512 and larger the mean GPU Computation Percentage ranged from 0.67 to 0.98. For device efficiencies near 0.5 the amount of time transferring data was about the same as the time spend doing computation. This GPU Computation Percentage value leads us to conclude that it would be more efficient to run problem sizes with GPU Computation Percentage near 0.5 on a single device. We need to look at wall clock time to confirm this. Comparing the GPU Computation Percentage of the tiled kernel and the tiled plus shared memory kernel, the GPU Computation Percentage was lower for the tiled plus shared memory kernel in all cases.

Comparing the GPU Computation Percentage for both tiled and tiled plus shared memory matrix multiplication benchmark using paged memory in Table 9 with the GPU Computation Percentage for the matrix multiplication benchmark using pinned memory in Table 10 we see tiled plus shared kernel on problem sizes 1024x1024 and larger show a noticeable difference in GPU Computation Percentage. This suggests data movement was decreasing.

Table 9: GPU Computation Percentage, paged memory two devices

matrix size	naive		tiled		tiled+shared	
	dev 0	dev 1	dev 0	dev 1	dev 0	dev 1
32x32	0.65	0.55	0.64	0.63	0.47	0.47
512x512	-		0.93	0.93	0.67	0.67
1024x1024	-		0.94	0.94	0.72	0.65
2048x2048	-		0.95	0.95	0.80	0.80
4096x4096	-		0.97	0.97	0.89	0.88
8192x8192	-		0.98	0.98	0.93	0.93
16384x16384	-		0.99	0.99	0.96	0.96

Table 10: GPU Computation Percentage, pinned memory two devices

matrix size	naive		tiled		tiled+shared	
	dev 0	dev 1	dev 0	dev 1	dev 0	dev 1
32x32	0.61	0.62	0.60	0.60	0.43	0.43
512x512	-		0.93	0.93	0.67	0.67
1024x1024	-		0.95	0.95	0.76	0.76
2048x2048	-		0.97	0.97	0.86	0.86
4096x4096	-		0.98	0.98	0.92	0.92
8192x8192	-		0.99	0.99	0.96	0.96
16384x16384	-		1.00	1.00	0.98	0.98

Table 11: GPU Computation Percentage, pinned memory, asynchronous memory transfers, two devices

matrix size	naive		tiled		tiled+shared	
	dev 0	dev 1	dev 0	dev 1	dev 0	dev 1
32x32	0.62	0.62	0.60	0.60	0.43	0.43
512x512	-		0.93	0.93	0.67	0.67
1024x1024	-		0.95	0.95	0.76	0.76
2048x2048	-		0.97	0.97	0.86	0.86
4096x4096	-		0.98	0.98	0.92	0.92
8192x8192	-		0.99	0.99	0.96	0.96
16384x16384	-		1.00	1.00	0.98	0.98

7.3. Hybrid Environment Results

Table 12 shows the mean GPU Computation Percentage for the matrix multiplication benchmark overlapping CPU and GPU computation. In Case A no memory optimization was used, in Case B pinned memory was used, and in Case C pinned memory with asynchronous memory transfers was used. The naive column shows the metric value for naive kernel, the tiled column shows the metric values for the tiled kernel, and the tiled+shared column shows the metric values for the tiled with shared memory kernel.

First we examine the naive kernel. The mean GPU Computation Percentage for all three memory optimizations was very similar, ranging from 0.61 to 0.65.

Next we examine the tiled kernel. The mean GCP for the 32x32 problem size ranged from 0.60 to 0.64. At this problem size it may be faster to do the computation strictly on the host. For problem sizes 512x512 and larger the mean GPU Computation Percentage ranged from 0.93 to 0.98. In all of these cases, the device spent more time doing work than was spent transferring data; therefore, the overhead of transferring data to the device was justified.

Examining the tiled plus shared memory kernel, the mean GPU Computation Percentage for the 16x16 problem size ranged from 0.43 to 0.47. For problem sizes 512x512 and larger the mean GPU Computation Percentage ranged from 0.67 to 0.92.

Comparing the GPU Computation Percentage of the tiled kernel and the tiled plus shared memory kernel, the GCP was lower for the tiled plus shared memory kernel in all cases. The low GCP for the tiled plus shared memory kernel on a 32x32 matrix suggest that it would be more efficient to run those problem sizes strictly on a device; in fact the wall clock times for these runs (1.2 seconds, 1.8 seconds, 1.8 seconds) indicate worsening performance as we increase optimizations.

Table 12: GPU Computation Percentage, hybrid

matrix size	naive			tiled			tiled+shared		
	A	B	C	A	B	C	A	B	C
32x32	0.65	0.61	0.62	0.64	0.60	0.60	0.47	0.43	0.43
512x512	-	-	-	0.93	0.93	0.93	0.67	0.67	0.67
1024x1024	-	-	-	0.95	0.95	0.95	0.75	0.77	0.77
2048x2048	-	-	-	0.96	0.97	0.97	0.82	0.86	0.86
4096x4096	-	-	-	0.98	0.98	0.98	0.90	0.92	0.92

7.4. Discussion

Our goal for the GPU Computation Percentage metric is to indicate the amount of data movement overhead between the CPU and GPU. Our results indicate that it can be used for this purpose. Looking at the results for a single optimization, e.g. tiled with paged memory, the GCP increases as the problem size increases. We expect this to happen with matrix multiplication because more computation is being done as the problem size increases. However, the metric falls short when comparing the GPU Computation Percentage of different kernel optimizations, e.g. tiled versus tiled plus shared memory, because the faster kernel has a lower GCP. The tiled plus shared memory optimization reduces the kernel device time, but the data movement time

stays the same. The decrease in GPU Computation Percentage is counter intuitive one might expect that the faster kernel would have a higher GPU Computation Percentage. Also, the GPU Computation Percentage metric doesn't definitively say that the cost of moving data to and from the device was justified. For example, a kernel may run extremely fast on the device and provide a significant performance improvement over the host, but still have a low GPU Computation Percentage because of how fast it executed on the device.

8. GPU Load Balance Case Study

Our goal for the GPU Load Balance metric is to show the load balance between the CPU and GPU or between multiple devices. The definition of GPU Load Balance (2) is shown below.

$$GLB = \frac{\text{device time}}{\text{wall clock time}} \tag{2}$$

A low GPU Load Balance indicates that the device was idle. A high GPU Load Balance indicates that the work was well balanced between the CPU and GPU or it was device centric.

8.1. Single Device Results

Table 13 to Table 15 show the mean GPU Load Balance for the matrix multiplication benchmark using a single device. For the data in Table 13 no memory optimization was used, in Table 14 pinned memory was used, and in Table 15 pinned memory with asynchronous memory transfers was used. The naive column shows the metric value for naive kernel, the tiled column shows the metric values for the tiled kernel, and the tiled+shared column shows the metric values for the tiled with shared memory kernel.

In some cases the mean GPU Load Balance in Table 13 to Table 15 is being reported as 0.00. A kernel was executed on the device for all the experiments, and we verified the kernel device times are non zero; however, the external time utility was used to measure wall clock time, and it only has a resolution of a hundredth of a second.

Therefore, we only have two decimal places of precision and these very small values are rounding to zero.

First we examine the naive kernel. The mean GPU Load Balance for all three memory optimizations was zero. This suggests that very little of the device's available computation time was used.

Next we examine the tiled kernel. The mean GPU Load Balance for the 16x16 to 1024x1024 problem sizes was near zero as well. Only for the 4096x4096 and larger problem sizes was the GPU Load Balance above 0.5. This suggests that the matrix multiplication benchmark was only device centric for these larger problem sizes.

For the tiled plus shared memory kernel, the GPU Load Balance was less than 0.5 for most of the 16x16 to 4096x4096 problem sizes. Comparing the GPU Load Balance of the tiled kernel and the tiled plus shared memory kernel, the GPU Load Balance was lower for the tiled plus shared memory kernel in most cases.

Comparing the GPU Load Balance for the matrix multiplication benchmark using paged memory, pinned memory, and pinned memory with asynchronous memory transfers we did not see a noticeable difference.

Table 13: GPU Load Balance, paged memory single device

matrix size	naive	tiled	tiled+shared
16x16	0.00	0.00	0.00
512x512	-	0.02	0.00
1024x1024	-	0.10	0.02
2048x2048	-	0.41	0.13
4096x4096	-	0.82	0.51
8192x8192	-	0.96	0.85
16384x16384	-	0.99	0.95

Table 14: GPU Load Balance, pinned memory single device

matrix size	naive	tiled	tiled+shared
16x16	0.00	0.00	0.00
512x512	-	0.01	0.00
1024x1024	-	0.06	0.01
2048x2048	-	0.31	0.09
4096x4096	-	0.74	0.38
8192x8192	-	0.94	0.78
16384x16384	-	0.99	0.93

Table 15: GPU Load Balance, pinned memory, asynchronous memory transfers, single device

matrix size	naive	tiled	tiled+shared
16x16	0.00	0.00	0.00
512x512	-	0.01	0.00
1024x1024	-	0.06	0.01
2048x2048	-	0.31	0.09
4096x4096	-	0.74	0.37
8192x8192	-	0.94	0.78
16384x16384	-	0.99	0.93

8.2. Multiple Device Results

Table 16 to Table 18 show the mean GPU Load Balance for the matrix multiplication benchmark using two devices. For the data in Table 16 no memory optimization was used, in Table 17 pinned memory was used, and in Table 18 pinned memory with asynchronous memory transfers was used. The naive column shows the metric value for naive kernel, the tiled column shows the metric values for the tiled kernel, and the tiled+shared column shows the metric values for the tiled with shared memory kernel.

In some cases the mean GPU Load Balance in Table 16 to Table 18 is being reported as 0.00. A kernel was executed on the device for all the experiments, and we verified the kernel device times are non zero; however, the external time utility was used to measure wall clock time, and it only has a resolution of a hundredth of a second. Therefore, we only have two decimal places of precision and these very small values are rounding to zero.

For the naive kernel the mean GPU Load Balance for all three memory optimizations were zero. The result for the tiled kernel run on two devices was similar to the results for the tiled kernel run on a single device. Comparing the results for the tiled plus shared memory kernel run on a single device and on two devices it was noticed that with two devices the kernel became device centric at problem sizes 8192x8192 instead of the 4096x4096 problem size. Comparing the GPU Load Balance of the tiled kernel and the tiled plus shared memory kernel, the GPU Load Balance was lower for the tiled plus shared memory kernel in all cases.

Table 16: GPU Load Balance, paged memory two devices

matrix size	naive		tiled		tiled+shared	
	dev 0	dev 1	dev 0	dev 1	dev 0	dev 1
32x32	0.00	0.00	0.00	0.00	0.00	0.00
512x512	-		0.01	0.01	0.00	0.00
1024x1024	-		0.05	0.05	0.01	0.01
2048x2048	-		0.25	0.25	0.08	0.08
4096x4096	-		0.69	0.69	0.32	0.32
8192x8192	-		0.93	0.93	0.69	0.69
16384x16384	-		0.98	0.98	0.91	0.91

Table 17: GPU Load Balance, pinned memory two devices

matrix size	naive		tiled		tiled+shared	
	dev 0	dev 1	dev 0	dev 1	dev 0	dev 1
32x32	0.00	0.00	0.00	0.00	0.00	0.00
512x512	-		0.01	0.01	0.00	0.00
1024x1024	-		0.03	0.03	0.01	0.01
2048x2048	-		0.17	0.17	0.04	0.04
4096x4096	-		0.56	0.56	0.23	0.23
8192x8192	-		0.88	0.88	0.61	0.61
16384x16384	-		0.97	0.97	0.85	0.85

Table 18: GPU Load Balance, pinned memory, asynchronous memory transfers, two devices

matrix size	naive		tiled		tiled+shared	
	dev 0	dev 1	dev 0	dev 1	dev 0	dev 1
32x32	0.00	0.00	0.00	0.00	0.00	0.00
512x512	-		0.01	0.01	0.00	0.00
1024x1024	-		0.03	0.03	0.01	0.01
2048x2048	-		0.17	0.17	0.04	0.04
4096x4096	-		0.56	0.56	0.23	0.23
8192x8192	-		0.88	0.88	0.61	0.61
16384x16384	-		0.97	0.97	0.85	0.85

8.3. Hybrid Environment Results

Table 19 to Table 21 show the mean GPU Load Balance for the matrix multiplication benchmark overlapping CPU and GPU computation. For the data in Table 19 no memory optimization was used, in Table 20 pinned memory was used, and in Table 21 pinned memory with asynchronous memory transfers was used. The naive column shows the metric value for naive kernel, the tiled column shows the metric values for the tiled kernel, and the tiled+shared column shows the metric values for the tiled with shared memory kernel.

In some cases the mean GPU Load Balance in Table 19 to Table 21 is being reported as 0.00. A kernel was executed on the device for all the experiments, and we verified the kernel device times are non zero; however, the external time utility was used to measure wall clock time, and it only has a resolution of a hundredth of a second. Therefore, we only have two decimal places of precision and these very small values are rounding to zero.

The mean GPU Load Balance for all problem sizes was near zero for all three kernels across all the memory optimizations. This is because the host execution dominated the execution time.

Table 19: GPU Load Balance, paged memory, hybrid

matrix size	naive	tiled	tiled+shared
32x32	0.00	0.00	0.00
512x512	-	0.01	0.00
1024x1024	-	0.01	0.00
2048x2048	-	0.01	0.00
4096x4096	-	0.01	0.00

Table 20: GPU Load Balance, pinned memory, hybrid

matrix size	naive	tiled	tiled+shared
32x32	0.00	0.00	0.00
512x512	-	0.01	0.00
1024x1024	-	0.01	0.00
2048x2048	-	0.01	0.00
4096x4096	-	0.01	0.00

Table 21: GPU Load Balance, pinned memory, asynchronous memory transfers, hybrid

matrix size	naive	tiled	tiled+shared
32x32	0.00	0.00	0.00
512x512	-	0.01	0.00
1024x1024	-	0.01	0.00
2048x2048	-	0.01	0.00
4096x4096	-	0.01	0.00

8.4. Discussion

Looking at the data in Table 13 to Table 18 it can be seen that the GPU Load Balance increases as the matrix size is increased. This is because more work is being performed on the device and device execution time is dominating the total execution time. For the small matrices the overhead to allocate memory and launch the kernel dominates the execution time. However, for the larger matrices, kernel execution dominates the execution time.

Looking at the data in Table 13 to Table 18 it can be seen that the tiled optimization has a higher GPU Load Balance than the tiled plus shared memory optimization. This is because the matrix multiplication application strictly uses the devices for calculations, and the tiled plus shared memory optimization decreases the amount of time spent executing on the device.

Comparing the single device versions with the two device versions it can be seen that the two device version has a lower GPU Load Balance. This is because the matrix multiplication is being divided among the devices, thus each device is being used less. The pinned memory versions also have lower GPU Load Balance than the paged memory version. This is because the memory transfers between the host and device are occurring faster, which decreases the device time reported by CudaProf for these memory transfers.

Comparing the pinned memory version with the pinned memory with asynchronous memory transfer version it can be seen they have nearly identical results. Investigating this it was realized that no additional streams were used. If no streams are created all memory transfers to and from the device occur on the default stream which causes them to be serialized. Thus, to see a performance improvement when using asynchronous memory transfers streams also need to utilized.

Our goal for the GPU Load Balance metric was to show the developer the load balance between the CPU and GPU or between multiple GPUs. The GPU Load Balance metric shows that the GPU Load Balance is increasing as the problem size increases. This is as expected because the matrix multiplication benchmark becomes more device centric as the problem size increases. The two device version of the matrix multiplication benchmark shows that the work was balanced between the devices. This is expected because we implemented it to evenly divide the work.

9. NAMD Case Study

In the previous case studies we explored the GPU Computation Percentage metric and GPU Load Balance metric using a matrix multiplication benchmark. In this case study we investigate these metrics using NAMD that utilizes GPUs to perform computations that are overlapped with computation on the host CPU.

NAMD was configured to simulate STMV. The simulation was run ten times using one GPU device and ten times using two GPU devices. The mean GCP is shown in Table 22.

Table 22: NAMD GPU Computation Percentage

single device	two devices	
dev 0	dev 0	dev 1
0.99	0.98	0.98

Since the GPU Computation Percentage was greater than 0.5 more time was spent doing computation on the device than moving data to the device. We found similar results for the tiled matrix multiplication kernel with problem sizes 512x512 and larger. We also found similar results for the tiled plus shared memory matrix multiplication kernel on most of the problem sizes 4096x4096 and larger. The high GPU Computation Percentage results indicate that the data movement overhead was low for NAMD.

The mean GPU Load Balance is shown in Table 23. The GPU Load Balance with one device was 0.44. When used with two devices the GPU Load Balance for NAMD

decreased to between 0.28 and 0.29. We found similar results for the tiled matrix multiplication kernel with paged memory on a single device at problem size 2048x2048, and for the tiled plus shared memory matrix multiplication kernel with paged memory on a single device at problem size 4096x4096.

Table 23: NAMD GPU Load Balance

single device	two devices	
dev 0	dev 0	dev 1
0.44	0.29	0.28

The GPU Load Balance for NAMD when run with two devices indicates that the work was nearly balanced between the two devices. However, the percentage of time the devices were used decreased from the single device simulation.

10. Conclusions and Future Work

We conducted this thesis work as a starting point to model the performance of hybrid CPU/GPU systems. To accomplish this we defined two metrics: GPU Computation Percentage and GPU Load Balance. Our goal for the GPU Computation Percentage metric is to indicate the amount of data movement overhead between the CPU and GPU. Our results indicate that GPU Computation Percentage can be used to show the data movement overhead. However, between optimizations of a kernel the GPU Computation Percentage decreases. This is counter intuitive in the sense that "better" yields a lower value, as does "worse."

Our goal for the GPU Load Balance metric is to indicate the load balance between the CPU and GPU or between multiple GPUs. We found that it can be used to indicate load balance. When used with a single device it indicated how device centric the matrix multiplication benchmark and NAMD were. When used with multiple devices it indicated how balanced the work was between the devices.

In the future we would like to investigate the metrics using a distributed memory system, with device to device memory transfers, and with different kernel configurations. Future work would also include implementing an optimized matrix multiplication benchmark for the host.

In PerfTrack we would like to develop an easier method for correlating performance results between GPU devices.

In order to get to get a complete view of CPU and GPU performance data we would like to instrument the CUDA runtime system directly. This approach would not be as portable and requires access to a proprietary code base; but would allow us to get a more accurate estimate of data movement overheads.

11. References

[1] L. Adhianto et al., "HPCToolkit: Tools for performance analysis of optimized parallel programs" in Concurrency and Computation: Practice and Experience, 2010, to be published.

[2] V. Allada et al., "Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster," in *IEEE International Conference on Cluster Computing and Workshops, 2009*, New Orleans, LA, 2009, pp. 1-9.

[3] S. Baghsorkhi, et al., "An adaptive performance modeling tool for GPU architectures," in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, Bangalore, India, 2010, pp. 105-114.

[4] A. Bakhoda, et al., "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Boston, MA, 2009, pp. 163-174.

[5] J. Brandt, et al., "Ovis-2: A robust distributed architecture for scalable RAS," in *IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, 2008, pp. 1-8.

[6] H. Brunst, et al., "Tools for Scalable Parallel Program Analysis - Vampir VNG and DeWiz," in *Distributed and Parallel Systems: Cluster and Grid Computing*, Z. Juhasz, P. Kacsuk and D. Kranzlmüller, Eds. Kluwer International Series in Engineering and Computer Science, New York, NY: Springer, pp. 93-102.

[7] P. Bui and J. Brockman, "Performance analysis of accelerated image registration using GPGPU," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, Washington, D.C., 2009, pp 38-45.

[8] S. Che, et al., "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization*, Austin, TX, 2009, pp. 44-54.

[9] Nvidia. (July 7, 2010). *CUDA_Profiler_3.0.txt* [Online]. Available: http://developer.nvidia.com/object/cuda_3_1_downloads.html.

[10] A. Danalis, et al., "The Scalable Heterogeneous Computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, Pittsburgh, PA, 2010, pp. 63-74, 2010.

[11] S. Dhawan, "Introduction to PCI Express-a new high speed serial data bus," in *2005 IEEE Nuclear Science Symposium Conference Record*, Fajardo, 2005, pp. 687-691.

[12] Z. Fan, et al., "GPU Cluster for High Performance Computing," in *Proceedings of the ACM/IEEE SC2004 Conference Supercomputing*, Pittsburgh, PA, 2004, pp. 47.

[13] Peter L. Freddolino, et al., "Molecular Dynamics Simulations of the Complete Satellite Tobacco Mosaic Virus", *Structure*, vol. 14, issue 3, pp. 437-449, Mar. 2006.

[14] M. Geimer, et al., "The Scalasca performance toolset architecture," in *Concurrency and Computation: Practice and Experience*, vol. 22, issue 6, pp. 702-719, Apr. 2010.
[15] Ubuntu man pages. (June 23, 2010). *gettimeofday()* [Online]. Available: http://manpages.ubuntu.com/manpages/jaunty/en/man2/gettimeofday.2.html.
[16] D. Goddeke, et al., "Exploring weak scalability for FEM calculations on a GPU-enhanced cluster," in *Parallel Computing*, vol. 33, issue 10-11, pp. 685-699, Nov. 2007.
[17] K. Govindaraju, et al., "High performance discrete Fourier transforms on graphics processors," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Austin, TX, 2008, pp. 1-12.
[18] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, Austin, TX, 2009, pg. 152-163.
[19] S. Huang, et al, "On the energy efficiency of graphics processing units for scientific computing," in *IEEE International Symposium on Parallel & Distributed Processing*, Rome, 2009, pp. 1-8.
[20] D. Jamsek and E. Van Hensbergen, "Experiences with hybrid clusters," in *IEEE International Conference on Cluster Computing and Workshops*, New Orleans, LA, 2009, pp. 1-4.
[21] K. Karavanic, et al., "Integrating Database Technology with Comparison-based Parallel Performance Diagnosis: The PerfTrack Performance Experiment Management Tool," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Seattle, WA, 2005, pp. 39.
[22] D. Kirk and W. Hwu. in *Programming Massively Parallel Processors*. Burlington, MA: Morgan Kaufmann Publishers, 2010.
[23] V. Kindratenko, et al., "GPU Clusters for High-Performance Computing," in *Workshop on Parallel Programming on Accelerator Clusters - PPAC'09*, New Orleans, LA, 2009, pp. 1-8.
[24] J. Kohn and W. Wiliams, "ATExpert," in *Journal of Parallel Distributed Computing*, vol. 18, issue 2, pp. 205-222, 1993.
[25] O. Lawlor, "Message passing for GPGPU clusters: CudaMPI," in *IEEE International Conference on Cluster Computing and Workshops*, New Orleans, LA, 2009, pp. 1-8.
[26] W. Liu, et al., "Performance Predictions for General-Purpose Computation on GPUs," in *International Conference on Parallel Processing*, Xi'an, 2007, pp. 50.
[27] A. Maloney, et al., "An Experimental Approach to Performance Measurement of Heterogeneous Parallel Applications using CUDA," in *Proceedings of the 24th ACM international Conference on Supercomputing*, Tsukuba, Ibaraki, Japan, 2010, pp. 127-136.
[28] C. Muller, et al., "A Compute Unified System Architecture for Graphics Clusters Incorporating Data Locality," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, issue 4, pp. 605-617, 2009.

[29] Nvidia. (February 2010). *NVIDIA CUDA Programming Guide* [Online]. Available: http://developer.nvidia.com/object/cuda_3_1_downloads.html.
[30] Nvidia. (June 28, 2010). *Nvidia Parallel Nsight* [Online]. Available: http://developer.nvidia.com/object/nsight.html.
[31] Parboil Benchmark Suite. (June 28, 2010). [Online]. Available: http://impact.crhc.illinois.edu/parboil.php.
[32] D. Pase and W. Williams, "A Performance Tool for The CRAY T3D," in *Workshop on Debugging and Performance Tuning*, 1994.
[33] James C. Phillips, et al., "Scalable molecular dynamics with NAMD," in *Journal of Computational Chemistry*, vol. 26, pp. 1781-1802, 2005.
[34] J. Phillips, et al., "Adapting a message-driven parallel application to GPU-accelerated clusters," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Austin, TX, 2008, pp. 1-9.
[35] D. Roeh, et al., "Accelerating cosmological data analysis with graphics processors," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, Washington, D.C., 2009, pp. 1-8.
[36] S. Ryoo, et al., "Program optimization space pruning for a multithreaded gpu," in *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, Boston, MA, 2008, pp. 195-204.
[37] D. Schaa, and D. Kaeli, "Exploring the multiple-GPU design space," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, 2009, pp. 1-12.
[38] S. Shende and A. Malony, "The TAU Parallel Performance System," in *International Journal of High Performance Computing Applications*, vol. 20, issue 2, pp. 287-331, May 2006.
[39] M. Showerman, et al., "QP: A heterogeneous multi-accelerator cluster," in *Proceedings of the 10th LCI International Conference on High-performance Clustered Computing*, Pittsburgh, PA, 2009.
[40] Ubuntu man pages. (June 23, 2010). *time command man page* [Online]. Available: http://manpages.ubuntu.com/manpages/jaunty/en/man1/time.1.html.
[41] G. Teodoro, et al., "Coordinating the use of GPU and CPU for improving performance of compute intensive applications," in *IEEE International Conference on Cluster Computing and Workshops*, New Orleans, LA, 2009, pp. 1-10.
[42] The Hybrid Multicore Consortium. (July 10, 2010). [Online]. Available: http://computing.ornl.gov/HMC/index.html
[43] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Austin, TX, 2008, pp. 1-11.
[44] S. Williams, et al., "Roofline: an insightful visual performance model for multicore architectures," in *Communications of the ACM*, vol. 52, issue 4, pp. 52, 65-76, Apr. 2009.
[45] S. Yamagiwa and K. Wada, "Performance study of interference on GPU and CPU resources with multiple applications," in *IEEE International Symposium on Parallel & Distributed Processing, 2009*, Rome, 2009, pp. 1-8.