2013-01-01

Efficient, Scalable, Parallel, Matrix-Matrix Multiplication

Enrique Portillo
University of Texas at El Paso, portillo11@gmail.com

EFFICIENT, SCALABLE, PARALLEL MATRIX-MATRIX MULTIPLICATION

ENRIQUE PORTILLO

Department of Computer Science

APPROVED:

Patricia J. Teller, Ph.D., Chair

Sarala Arunagiri, Ph.D., Co-Chair

Shirley Moore, Ph.D.

Soheil Nazarian, Ph.D.

Benjamin C. Flores, Ph.D.
Dean of the Graduate School

Dedication

- To my mother for her unconditional support and love; it is because of you I know how to love.
- To my father for teaching me to aim high and be the best I can be; it is because of you I have no limits.
- To my sister for making my life intese; it is because of you I'm passionate.
- To my brother for teaching me that the real learning starts when you are the most exhausted.
- And finally to Romeo for making our efforts worthwhile; it is in your hands to lead our next generation.
- Every achievement I shall receive is due to all of you.

EFFICIENT, SCALABLE, PARALLEL MATRIX-MATRIX MULTIPLICATION

by

Enrique Portillo, B.S.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

December 2013

Acknowledgements

I wish to extend my absolute gratitude to my advisors, Doctor Patricia Teller and Sarala Arunagiri. You two teach me to appreciate the art in Computer Science and gave me strength and direction when I most needed it. Sarala's patience is unbeatable and her ideas bright. Dr. Teller merged my professional path to my life path, and I can't be enough grateful for such unconditional support.

Abstract

For the past decade, power/energy consumption has become a limiting factor for large-scale and embedded High Performance Computing (HPC) systems. This is especially true for systems that include accelerators, e.g., high-end computing devices, such as Graphics Processing Units (GPUs), with terascale computing capabilities and high power draws that greatly surpass that of multi-core CPUs. Accordingly, improving the node-level power/energy efficiency of an application can have a direct and positive impact on both classes of HPC systems.

The research reported in this thesis explores the use of software techniques to enhance the execution-time and power-consumption performance of applications executed on a CPU/GPGPU compute node. We conducted this exploration in the context of parallel matrix-matrix multiplication with the goal of designing and developing a Single-precision General Matrix-matrix Multiplication (SGEMM) routine for CPU/GPGPU node execution that executes faster and/or consumes less power than competing routines. This thesis reports on this work, which resulted in an efficient, scalable, parallel single-precision matrix-matrix multiplication routine for square matrices that has comparable performance to existing routines but can multiply larger matrices and is limited by the size of the host CPU memory, instead of the size of the GPGPU device memory.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

For the past decade, power/energy consumption has become a limiting factor for large-scale and embedded High Performance Computing (HPC) systems. Currently, large-scale systems consume vast amounts of energy to provide for the high demand of state-of-the-art computational science and engineering applications [3] [10] [12] [19]. Furthermore, the power and cooling costs of these large-scale systems are becoming on par with acquisition costs [10] [17].

In large-scale computing clusters, a small improvement in energy efficiency at the node level can translate to substantial power savings. This is more common now that advances in technology have enabled the creation of high-end computing devices, such as Graphics Processing Units (GPUs), with terascale computing capabilities that can be used in embedded systems [8][16] as well as large-scale HPC systems. GPUs have a high power quota to meet and due to their compute capabilities their overall power draw can greatly surpass that of a multi-core CPUs. This results in an increasing demand on power, a limited resource in embedded systems [16].

In order to increase the power/energy efficiency of HPC systems and applications, it is critical to analyze and measure the power consumption of real HPC systems and applications at fine granularity [3] [8] [10]. Nonetheless, it is challenging to increase the power/energy efficiency of an application without decreasing its execution-time performance, i.e., application power/energy and execution-time performance are strongly connected, thus, modifying one will affect the other [8] [10]. In summary, improving the power/energy efficiency of an application can have a direct and positive impact on both classes of HPC systems, i.e., both large-scale and embedded systems.

The research reported in this thesis explores the use of software techniques to enhance the execution-time and power-consumption performance of applications executed on a CPU/GPGPU compute node. We conducted this exploration in the context of parallel matrix-matrix multiplication with the goal of designing and developing a Single-precision General Matrix-matrix Multiplication

(SGEMM) routine for CPU/GPGPU node execution that executes faster and/or consumes less power than competing routines. This thesis reports on this work, which resulted in an efficient, scalable, parallel single-precision matrix-matrix multiplication routine matrices (1) with execution-time and power performance that is comparable to CUBLAS and CUSUMMA, (2) that can multiply larger matrices than these routines, and (3) is limited by the size of the host CPU memory, instead of the size of the GPGPU device memory, which is the limiting factor for these routines.

1.1 BACKGROUND

The following section provides background information that is meant to familiarize the reader with the motivation for this research. This is followed by Section 1.2, which describes the contributions of this thesis and Section 1.3, which provides the organization of the thesis.

1.1.1 GPGPU

General-Purpose Graphics Processing Units (GPGPU), also known as *GPU computing,* is a term coined by Mike Harris in 2002 [15] when he noticed an early tendency to use GPUs for non-graphic applications. GPUs are high-performance accelerators, based on a many-core architecture, that are capable of very high computation and data throughput. In the early years, GPUs where designed specifically for computer graphics and were challenging to program. Nowadays, GPUs are general-purpose parallel co-processors, i.e., GPGPUs, which support industry-standard languages like C and accessible programming interfaces such as NVIDIA CUDA. Scientific workloads ported to GPGPUs frequently attain orders of magnitude of speedup compared to optimized CPU implementations [1] [2] [9] [11] [18] [20] [23].

Accelerators are now common place in HPC systems. More specifically, GPGPU-based accelerators have been well adopted by energy-efficient computer systems, e.g., the first eight most energy-efficient supercomputers in the TheGreen500 List [12] are accelerator-based architectures. Although GPGPUs are power hungry (e.g., they can consume half the power of the system), they also

provide much more computational (multiple orders of magnitude) throughput than do current multi-core CPUs (i.e., for embarrassingly parallel workloads) [9]. Because of this fact, a host CPU paired with a GPGPU has the potential to attain better performance-per-watt than less power hungry individual multi-core CPUs [7]. Towards this end, the research associated with this thesis explored the use of execution-time and power/energy consumption techniques in the context of CPU/GPGPU matrix multiplication.

1.2.2 Matrix Multiplication

Single-precision General Matrix-Matrix Multiplication (SGEMM), both dense and sparse, is an important application kernel, and it is omnipresent in scientific computing [4] [9] [22] [23]. This kernel is essential to numerous linear algebra numerical algorithms, computational models of biological neural networks, computer graphics, linear discrete dynamical systems, quantum mechanics, and more [5] [9] [14].

Due to its regular access patterns, the SGEMM kernel maps to a Single Instruction Multiple Data (SIMD) architecture – that of GPGPUs – in a trivial manner. Parallelization of SGEMM is uncomplicated since input matrices A and B, and output matrix C can be divided into submatrices that can be operated on by different processes. SGEMM has $O(N^3)$ compute complexity and $O(N^2)$ data access complexity, where N is the matrix dimension, i.e., the matrices are of size $N \; x \; N$ [14]. Due to this characteristic, SGEMM is classified as a compute-bound application.

There are two elementary methods to compute the product of two matrices, i.e., inner and outer product. Given input matrices A and B, and output matrix C of sizes $m \; x \; k$, $k \; x \; n$, and $m \; x \; n$, respectively, the inner-product method works by computing the dot product of each row vector of A and each column vector of B, which generates a single result element of matrix C. For example, the dot product of the elements in row 3 of A and the elements of column 2 of B produces the element $C_{3,2}$ in the result matrix C. In contrast, the outer-product method generates $k \; m \; x \; n$ submatrices of partial results, which are added together to form the final result matrix C. The submatrix i ($i = 1, \ldots, k$) is generated using

column vector *i* of A and row vector *i* of B. The resultant $m\ x\ n$ submatrix is created by computing the tensor product of the column of A and the row of B. To better understand the differences between these two approaches, we present an example that depicts each approach for multiplying a 3 x 3 matrix A and a 3 x 2 matrix B, which results in a 3 x 2 result matrix C.

Given

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}, B = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix},$$

the inner-product method to multiply $A\ x\ B$ results in:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} x \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} = \begin{pmatrix} (a\ x\ 1) + (b\ x\ 2) + (c\ x\ 3) & (a\ x\ 4) + (b\ x\ 5) + (c\ x\ 6) \\ (d\ x\ 1) + (e\ x\ 2) + (f\ x\ 3) & (d\ x\ 4) + (e\ x\ 5) + (f\ x\ 6) \\ (g\ x\ 1) + (h\ x\ 2) + (i\ x\ 3) & (g\ x\ 4) + (h\ x\ 5) + (i\ x\ 6) \end{pmatrix}$$

in contrast, the outer-product method results in:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} x \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} = \begin{pmatrix} (a\ x\ 1) & (a\ x\ 4) \\ (d\ x\ 1) & (d\ x\ 4) \\ (g\ x\ 1) & (g\ x\ 4) \end{pmatrix} + \begin{pmatrix} (b\ x\ 2) & (b\ x\ 5) \\ (e\ x\ 2) & (e\ x\ 5) \\ (h\ x\ 2) & (h\ x\ 5) \end{pmatrix} + \begin{pmatrix} (c\ x\ 3) & (c\ x\ 6) \\ (f\ x\ 3) & (f\ x\ 6) \\ (i\ x\ 3) & (i\ x\ 6) \end{pmatrix}.$$

1.3 CONTRIBUTIONS

This thesis reports on the research that was done to design and develop AHPCRCMM, an efficient, scalable, parallel matrix-matrix multiplication routine for execution on a CPU/GPGPU node. AHPCRCMM was designed to provide better execution-time and power/energy performance than existing routines used for this purpose. An experimental study was conducted to quantitatively evaluate the performance of AHPCRCMM, as compared to that of CUBLAS and CUSUMMA, the two most well-known routines of this type. To conduct this study, a test-bed was designed and developed to measure the power consumption of the CPU, GPU, RAM, and total system. Thus, the main products of this research include the following:

1. Design and development of the AHPCRCMM_Paged and AHPCRCMM_Pinned, two efficient, scalable, parallel matrix-matrix multiplication routines for execution on CPU/GPGPU nodes,

which (a) have execution-time and power performance that is comparable to CUBLAS and CUSUMMA, (2) can multiply larger matrices than these routines, and (3) are limited by the size of the host CPU memory, instead of the size of the GPGPU device memory, which is the limiting factor for these routines.

2. Design and development of a test-bed that was instrumented to measure the power consumption of multiple components of the computer system i.e., the power consumption of the CPU, RAM, and GPGPU, as well as total system power.
3. A comprehensive performance evaluation of the efficacy of AHPCRCMM as compared to two efficient parallel SGEMM methodologies, CUBLAS and CUSUMMA, which also is scalable and oriented to low data transfer between host and GPGPU device.

1.4 ORGANIZATION

The remainder of this thesis is organized as follows. Chapter 2 describes related work, focusing on parallel matrix multiplication for clusters and CPU/GPU nodes, and both execution-time and power performance of workloads executed on CPU/GPGPU nodes. Next, Chapter 3 describes our AHPCRCMM methodology in depth. The performance of AHPCRCMM is evaluated through an experimental methodology presented in Chapter 4, which includes the description of the test-beds, hardware and software tools, and execution-time performance and power measurements used in this research. In Chapter 5 we present and discuss our experimental results and compare the power/energy efficiency and execution-time performance of the three methodologies studied, i.e., the CUBLAS, CUSUMMA, and AHPCRCMM routines. Concluding remarks and future work are presented in Chapter 6, and finally complete code listings for our AHPCRCMM routine as well as additional power/energy and execution-time performance data are provided in the appendices.

Chapter 2: Related Work

As mentioned in Chapter 1, as part of this thesis, we designed and developed a single-precision matrix-matrix multiplication routine that executes on a CPU/GPGPU node to accelerate matrix-matrix multiplication. In this chapter we discuss related research. We focus on research in two areas: (1) parallel matrix-matrix multiplication for GPGPUs and (2) execution-time performance and power consumption of parallel workloads executed on CPU/GPGPU nodes.

Because we focus on node performance and matrix multiplication that is executed by a CPU/GPGPU pair, there are two competing routines, i.e., CUBLAS and CUSUMMA, to consider. Thus, the related work presented in this chapter focuses mainly on CUBLAS, CUSUMMA, and related efforts, as well as associated GPGPU performance studies. In Section 2.1 we present the work of others that produced parallel matrix-matrix multiplication methodologies, including CUBLAS and CUSUMMA. Section 2.2 focuses on research on execution-time and power performance measurements of applications executed on CPU/GPGPU nodes.

2.1 Parallel Matrix Multiplication for Clusters

Many efficient parallel matrix-matrix multiplication methods have been introduced for use in distributed-memory computer systems. In this section, we focus on two particular methods, PUMMA and SUMMA, which are most related to our research.

The block-cyclic approach used by Parallel Universal Matrix Multiplication Algorithms (PUMMA) [4] blocks (partitions) the input matrices into submatrices and then distributes them across available processing elements (nodes) that comprise a distributed-memory computer system, e.g., a cluster comprised of multiple nodes. Each node computes a partial result using an allocated submatrix and then moves that submatrix to another node, where it is used to compute another partial result. This sequence continues until all partial results are computed and the result matrix is complete. PUMMA is based on the inner-product method (explained in Chapter 1); it partitions and distributes submatrices

(across nodes) based on rows of matrix A and columns of matrix B. Furthermore, this methodology requires that the total number of submatrices be equal to the total number of processors, so that each submatrix pair, i.e., a submatrix of input matrices A and B, are mapped to each processor. This design strategy was chosen to maintain a good load balance across all processors.

In contrast, the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [22] is based on the outer-product method; it partitions and distributes submatrices across nodes based on columns of matrix A and rows of matrix B. Furthermore, this methodology requires that all the nodes of the parallel computer form a $r\ x\ c$ mesh of nodes, which is used as a logical mesh to map the submatrices of decomposed input matrices A and B. The main advantage of SUMMA, as compared to PUMMA, is its greater flexibility in the partitioning of the input matrices, i.e., SUMMA does not depend on the total number of processors but only on the total number of nodes. Furthermore, SUMMA partitions matrices over the shared dimension k of both input matrices A ($m\ x\ k$) and B ($k\ x\ n$), as opposed to PUMMA, which partitions dimensions m and n of each matrix, which results in a smaller number of data transfers between host and nodes.

2.2 Matrix Multiplication for CPU/GPU Nodes

As mentioned earlier, currently CUBLAS and CUSUMMA are the best known routines for multiplying matrices on a CPU/GPU node. Each is described below in the next two subsections.

2.2.1 CUBLAS

NVIDIA CUBLAS is a library of parallelized linear algebra subroutines for execution on NVIDIA CUDA-capable GPGPUs. The name CUBLAS comes from BLAS, which is a library of Basic Linear Algebra Subroutines, and CU, which stands for CUDA, the language used to program NVIDIA GPGPUs. All CUBLAS subroutines are required to store matrices and/or vectors in GPGPU device memory [15].

One of the subroutines in this library is the CUDA implementation of the BLAS SGEMM subroutine, which we refer to as the SGEMM kernel. In addition, this library provides a routine, written in C and CUDA and compiled by the NVIDIA NVCC compiler, which orchestrates the multiplication of matrices using the SGEMM kernel. This routine performs the following steps to multiply two matrices:

1. Allocate host memory for input matrices A and B, and output matrix C.
2. Populate A and B in host memory, and allocate GPGPU memory for all three matrices.
3. Transfer A and B from host memory to GPGPU memory.
4. Multiply A and B using the SGEMM kernel.
5. Transfer matrix C from GPGPU memory to host memory.
6. Free the host and GPGPU memory allocated to A, B, and C.

Note that the allocation and population of the memory for A and B is likely not necessary in real applications. Nonetheless, we do this in all the codes we use for comparison.

Because this routine requires that all matrices be stored in GPGPU memory, the matrix sizes that it can multiply are bounded by the GPGPU memory size, which usually is a fraction of the size of the host main memory. Thus, this routine can only be used in a subset of the experiments conducted as part of this research; we group this set of experiments together and call them the Class 1 experiments. Also, for the remainder of this thesis, we refer to the routine described above as CUBLAS.

2.2.2 CUSUMMA

Galbraith [9] extended the Scalable Universal Matrix Multiply Algorithm (SUMMA) to execute on a single CPU/GPGPU node, the GPGPU of which is an NVIDIA CUDA-capable GPGPU. This effort resulted in CUDA SUMMA (CUSUMMA), which is deemed to be scalable and portable. CUSUMMA dynamically partitions matrices based on GPGPU device memory capacity and matrix sizes [9]. Furthermore, CUSUMMA utilizes the CUBLAS SGEMM kernel to compute total or partial (depending on the matrix sizes) matrix multiplication results on the GPGPU device. In Galbraith's

research, the performance of CUSUMMA is compared to that of a purely serial implementation of the ATLAS-tuned BLAS SGEMM [5]. As expected, CUSUMMA achieved up to 12x speedup.

The matrix partitioning methodology of CUSUMMA focuses on minimizing the amount of data transferred between host and GPGPU device. Figure 2.1 illustrates CUSUMMA's partitioning strategies used for matrix sizes associated with Class 1, Class 2, and Class 3 experiments.

For Class 1 matrices, which are assumed to fit together in GPGPU memory, CUSUMMA uses the same six-step process used by CUBLAS, which was presented in the previous section. In essence, it transfers input matrices A and B from host to GPGPU memory, and stores matrix C in GPGPU memory. Class 2 matrices, which do not all fit together in GPGPU memory, are of sizes such that at least output matrix C is stored in GPGPU memory, along with blocks of matrices A and B. In general, for Class 2 matrices, CUSUMMA uses an outer-product approach to partition input matrices A and B, and then iteratively transfers blocks of A and B from host to GPGPU memory to permit the GPGPU to produce partial results and add them to matrix C in GPGPU memory. CUSUMMA performs the following steps to accomplish this:

1. Allocate host memory for input matrices A and B, and output matrix C.
2. Populate A and B in host memory, and allocate GPGPU memory for C and blocks of A and B.
3. Partition A and B at the host using the method described below.
4. Allocate host memory to one block of A; then copy the required elements from matrix A to this block. This is necessary because the elements of a column vector are not stored contiguously in host memory.
5. Transfer a block of A and the required elements from matrix B to GPGPU memory, and free the host memory associated with the block of A.
6. Execute the SGEMM kernel and store the partial result in C (in GPGPU memory).
7. Repeat steps 4-6 until all blocks from A and B are processed.

8. Transfer result matrix C from GPGPU memory to host memory.
9. Free the host and GPGPU memory allocated with A, B, and C.

Steps 3-7 are illustrated in Figure 2.1 for a pair of input matrices in Class 2: The input matrices of dimensions $m\ x\ k$ and $k\ x\ n$ are partitioned in dimension k to create two blocks of each matrix, i.e., blocks of dimensions $m\ x\ k1$ and $m\ x\ k2$ of the first matrix denoted as submatrix 1 and 2, respectively, and blocks of dimensions $k1\ x\ n$ and $k2\ x\ n$ of the second matrix, denoted as submatrix A and B, respectively. These submatrices are used to generate partial results in two $m\ x\ n$ matrices, 1A and 2B, which are summed to produce matrix C (in GPGPU memory), which is then transferred to host memory.

Class 3 matrices are of sizes such that no entire matrix fits in GPGPU memory. Thus, in this case, CUSUMMA partitions rows and columns of the first ($m\ x\ k$) input matrix into $m_1, m_2, \cdots, m_T$ and $k_1, k_2, \cdots, k_D$ to create blocks of size $m_i\ x\ k_j$; the number of partitions depends on the size of the matrices. Also, dimension k of the second input matrix is partitioned using the same partitioning scheme used for the dimension k of first input matrix to create blocks of size $k_j\ x\ n$. Subsequently, the result matrix ($m\ x\ n$) is divided into blocks of size $m_i\ x\ n$, where the m_i dimension is equal to the m_i dimension of a block of the first input matrix. Figure 2.1 provides an example of the partitioning used for Class 3 matrices. The first input matrix is partitioned once in each dimension, i.e., m and k. Subsequently the second input matrix and the output matrix are partitioned once in dimensions k and n, respectively. Then following an inner-product method, CUSUMMA multiplies each block in a row of blocks of the first input matrix. First, blocks 1 and 2 are multiplied by blocks A and B to produce partial product submatrices 1A and 2B, respectively, which are summed in GPGPU device memory. This partial sum ($1A + 2B$) is copied to host memory. This process is repeated for the second row of the input matrix, i.e., blocks 3 and 4, generating partial results 3A and 4B, which are added in GPGPU memory and transferred to host memory to complete the final result. It is important to notice that all

blocks of the second matrix i.e., submatrices of B, were transferred twice from host memory to GPGPU memory.

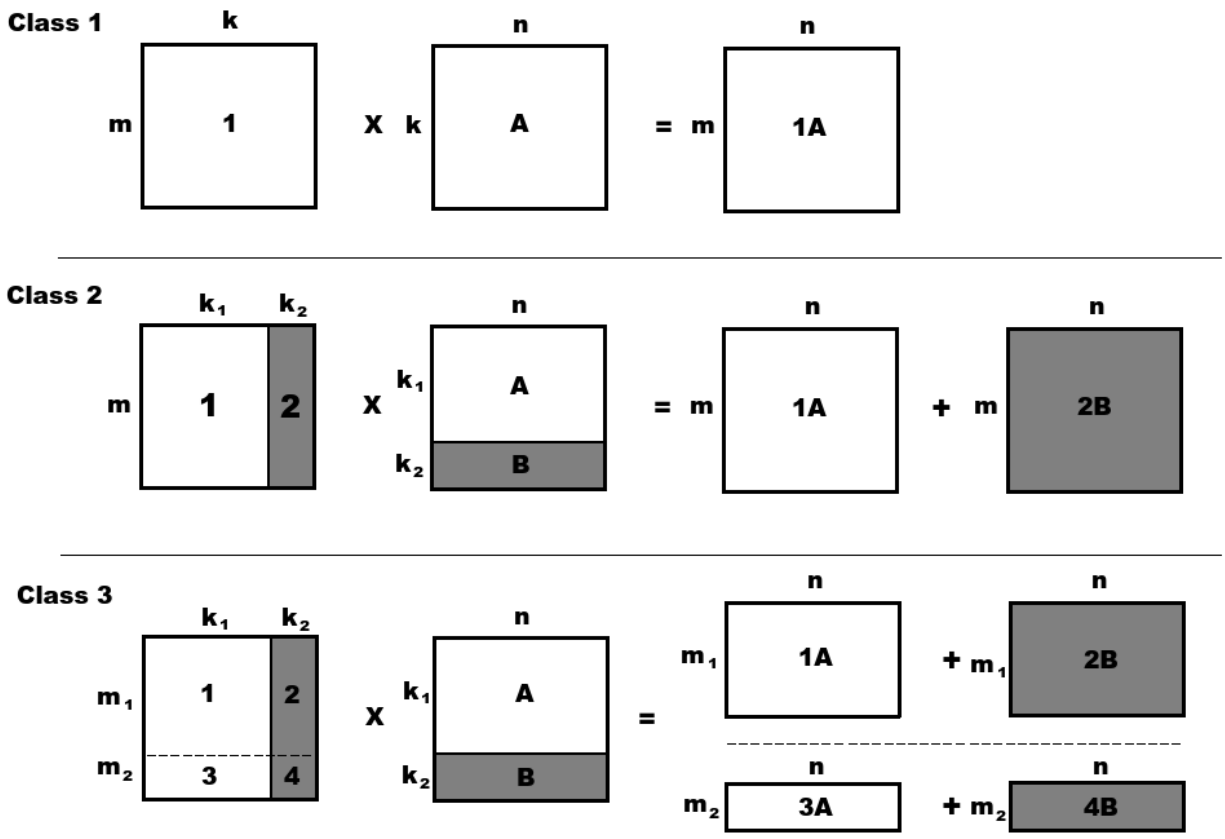

Figure 2.1: Partitioning strategy of CUSUMMA for each matrix class.

2.3 Execution-Time and Power Performance

In this section we discuss previous and current research related to both the execution-time and power performance of workloads executed on CPU/GPGPU nodes. There is a lot of work that is somewhat related to ours. We focus on two papers that are closest to our research. First, we review a study that measures the execution-time and power performance of common GPGPU workloads. Then, because CPU/GPGPU work-sharing is employed in one of the execution modes of AHPCRCMM, we summarize a performance study that focuses on the impact of work-sharing by the CPU and GPGPU of a compute node. For both, we differentiate our research from theirs.

2.3.1 Performance of Common GPU Workloads

Our research certainly is not the first to measure the execution-time and power-consumption performance of a set of GPGPU workloads. For example, Jiao, et al. [12] systematically characterized the energy efficiency of three common GPGPU workloads with different magnitudes of compute and memory intensiveness. More specifically, they investigated the execution-time performance and power consumption of these workloads executed under various core and memory clock frequencies. Based on their experimental results, they found that an application's execution time and power consumption have a strong correlation to: (1) the rate at which instructions are issued and (2) the ratio of the number of global memory transactions to the number of computation instructions.

Although we study GPGPU execution-time and power performance, we do so for a set of matrix multiplication routines executed on a CPU/GPGPU node with its default clock frequency. Also, unlike us, Jiao, et al. only consider total system power consumption, whereas we consider the power consumption of individual system components, i.e., CPU, GPGPU, and RAM, in addition to total system power. Finally, the measurement granularity of our power meter is five times smaller than the one used by Jiao, et al., thus, providing us with more accurate measurements.

2.3.2 Performance of CPU/GPU Work-Sharing

Our research also is not the first to investigate work-sharing between the CPU and GPGPU, with the purpose of reducing overall power consumption and/or execution-time performance. Qi Ren and Suda [20] investigated the power efficiency of work-sharing between a multi-core CPU and GPGPU co-processor. Their experimental results show that for the studied applications, sharing the execution of a workload reduces execution-time performance by more than 20% and provides an overall decrease in energy consumption of up to 6.17%. It should be noted that Qi Ren and Suda only take into account the power consumption of the CPU, which is derived from approximations of input current and voltage, and of the GPGPU, which was measured with an intrusive clamp probe on the auxiliary power

lines and an approximation derived from main board power inputs. In contrast, in our research we used two tools (described in Chapter 4) with better accuracy: (1) PAPI with RAPL enabled to acquire power consumption data directly from RAPL's MSR registers and (2) the NVIDIA-SMI application, which queries the GPGPU for its current power consumption state.

Chapter 3: AHPCRCMM

In this chapter we present AHPCRCMM, a single-precision matrix multiplication routine that we designed and developed to execute on a CPU/GPGPU node to accelerate matrix multiplication. AHPCRCMM was developed with the goal of outperforming, in terms of both execution time and power consumption, the two SGEMM routines that were developed for CPU/GPGPU use. These two routines, which were described in Chapter 2, are the widely used NVIDIA CUBLAS (which we refer to as CUBLAS) and CUSUMMA, which is an open-source project that extends the size of matrices that can be multiplied, in comparison to CUBLAS, via the use of matrix partitioning on the host CPU. AHPCRCMM was implemented in two ways, one using paged CPU host memory (AHPCRCMM_Paged) and the other using pinned CPU host memory (AHPCRCMM_Pinned). Although this routine could be modified easily to work on general matrices, in this thesis we focus on square matrices.

Section 3.1 describes the two solvers that AHPCRCMM uses to process input square matrices A and B, and output matrix C, prior to and after CUDA SGEMM kernel GPGPU execution. The selection of the solver to be used, AHPCRCMM Solver_1 or AHPCRCMM Solver_2, depends on the input matrix size and GPGPU memory capacity. Section 3.1 also provides the CPU and GPGPU memory requirements for the solvers, describes how data is transferred between the host and GPGPU, and indicates the techniques used to enhance execution-time and power performance. Since the same performance-enhancing techniques may be used by more than one solver, each technique is described separately in Section 3.2. Finally, in Section 3.3 the AHPCRCMM solvers are differentiated from CUSUMMA, in particular in terms of their efficient host-to-device memory transfers.

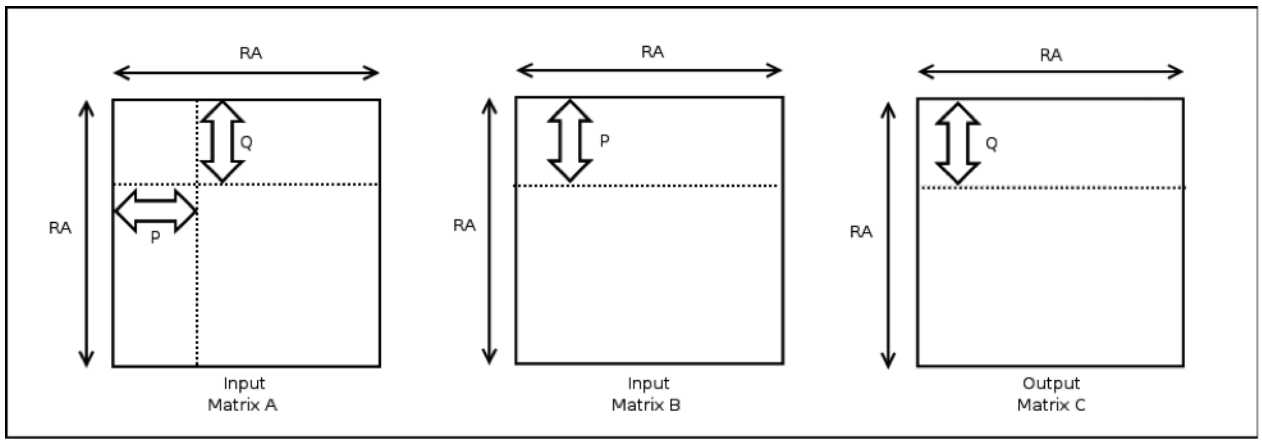

Figure 3.1: AHPCRCMM matrix partitioning

Table 3.1: Matrix parameters

Matrix Parameters	
RA	Number of elements in a dimension
P (matrix A)	Number of columns in block of A
P (matrix B)	Number of rows in block of B
Q (matrix A)	Number of rows in block of A
Q (matrix C)	Number of columns in block of C

3.1 MATRIX PROCESSING

Executing on a CPU/GPGPU pair, AHPCRCMM Solver_1 and AHPCRCMM Solver 2 perform multiplication of input matrices A and B to produce output matrix C. AHPCRCMM Solver_1 is used to multiply: (1) Class 1 matrices, i.e., matrices with sizes such that A, B, and C together fit in GPGPU device memory, which is a constraint of CUBLAS, and (2) Class 2 matrices, i.e., matrices with sizes such that matrix C fits in at most 96% of GPGPU device memory. When neither of these conditions holds and the capacity of the CPU host memory is sufficient to store four matrices of the specified size,

AHPCRCMM Solver_2 is adopted. AHPCRCMM Solver_2 multiplies what we call Class 3 matrices. Both Solver_1 when multiplying Class 2 matrices and Solver_2 perform matrix multiplication using the outer-product method described in Chapter 1. In contrast, CUSUMMA uses the outer-product method for the two smallest Class 2 matrices and the inner-product method for the rest of Class 2 matrices as well as Class 3 matrices.

AHPCRCMM, as well as CUSUMMA, can multiply matrices larger than those that can be multiplied by CUBLAS. And, AHPCRCMM can multiply matrices larger than those that can be multiplied by CUSUMMA since its constraints are associated with larger CPU host memory rather than GPGPU memory. When all three matrices do not fit in GPGPU memory, AHPCRCMM Solver_1 and Solver_2, as well as CUSUMMA partition matrices at the host CPU, transferring blocks to the GPGPU for multiplication. Regardless of their differences, AHPCRCMM, CUBLAS, and CUSUMMA always employ the CUDA SGEMM kernel, which was described in Chapter 2, to perform single-precision matrix-matrix multiplication on the GPGPU, whether multiplying full matrices or submatrices.

To help describe how AHPCRCMM Solver_1 and Solver_2 process matrices A, B, and C, Table 3.1 defines AHPCRCMM's matrix partitioning parameters and Figure 3.1 illustrates its partitioning of the matrices. Note that although this partitioning is applicable to general matrices, since this thesis focuses only on square matrices, it is explained below in terms of only square matrices. Each matrix is of size $RA \; x \; RA$; a block of A is a $Q \; x \; P$ submatrix, a block of B is a $P \; x \; RA$ submatrix, and a block of C is a $Q \; x \; RA$ submatrix. The number of elements in the matrices and the capacity of GPGPU device memory determine the values of P and Q. Moreover, a block from each matrix, even if it just one element, is required to fit in GPGPU memory at the same time.

3.1.1 AHPCRCMM Solver_1

As mentioned above, AHPCRCMM Solver_1 is employed to perform multiplication of square matrices when:

- **Condition 1**: Input matrices A and B, and output matrix C together fit in memory or
- **Condition 2**: Condition 1 does not hold and C fits in at most 96% of GPGPU memory.

The 96% threshold was determined by the following experiment: The size of matrix C, in comparison to the size of GPGPU memory, was increased in one-percent increments from 90% to 98% of GPGPU memory. For each matrix size, AHPCRCMM Solver_1 was executed and the matrix size that was associated with a drop in performance was used to identify the matrix size of choice, which was one that fits in at most 96% of GPGPU memory. A similar approach was used by Galbrait [9] to find the maximum amount of GPGPU memory that is allocated to matrix C by CUSUMMA. When Condition 1 holds, the following process, which is the same for CUBLAS and CUSMMA, is employed by Solver_1:

1. Allocate host memory for input matrices A and B, and output matrix C.
2. Populate A and B in host memory, and allocate GPGPU memory for all three matrices.
3. Transfer A and B from host memory to GPGPU memory.
4. Multiply A and B using the SGEMM kernel.
5. Transfer matrix C from GPGPU memory to host memory.
6. Free host and GPGPU memory allocated to A, B, and C.

When Condition 2 holds, the following process is employed by Solver_1:

1. Allocate host memory for input matrices A and B, and output matrix C.
2. Populate A and B in host memory, and allocate GPGPU memory for all three matrices.
3. Partition A and B at the host using the method described below.
4. Allocate GPGPU memory to one block of A, one block of B, and all of C with all elements initialized to zero; then copy the required elements of A and B to these blocks.

5. Transfer a block of matrix B ($P\ x\ RA$ elements) from host memory to GPGPU memory and free block of B at the host.
6. Transfer a block of matrix A ($RA\ x\ P$ elements) from host memory to GPGPU memory and free block of A at the host.
7. Multiply the blocks of A and B in GPGPU memory using the SGEMM kernel and add the partial results to matrix C in GPGPU memory.
8. Repeat steps 4-7 until all $\frac{RA}{P}$ blocks of A and B are processed.
9. Transfer matrix C from GPGPU memory to host memory.
10. Free host and GPGPU memory allocated to A, B, and C.

Employing this process, for each block (P rows) of B transferred to GPGPU memory, a block (P columns) of A is transferred and multiplied. Partial results are aggregated during this process, producing the final results matrix C.

Matrix Partitioning: For Solver_1, parameter Q is equal to RA and, thus, a block of C is the entire $RA\ x\ RA$ matrix, which occupies at most 96% of GPGPU memory. The remainder of GPGPU memory is needed to store a submatrix of matrix A of size $RA\ x\ P$ and a submatrix of matrix B of size $P\ x\ RA$. The value of P depends on the available GPGPU memory that remains after the allocation of matrix C. P has a minimum value of 1 (one column of A and one row of B) for the paged version and 2 for the pinned version (two columns of A and two rows of B). (It has a maximum value of RA when all matrices fit in GPGPU memory, i.e., when Condition 1 holds.) Since Solver_1 assumes that $Q = RA$ and, thus, C is stored in GPGPU memory, and the matrices it multiplies are square, it calculates the value of P as follows:

1. Compute *EGMS* (the Effective Global Memory Space), the total number of single-precision floating-point elements that fits in GPGPU device memory.

2. Compute $Celements = RA\ x\ Q = RA\ x\ RA$; *Celements* is the total number of floating-elements in matrix C.
3. Compute $Aelements = EGMS - Celements$; *Aelements* is the number of elements that can fit in the remainder of GPGPU memory.
4. Compute $P = \frac{Aelements}{RA+RA}$. Note that $RA + RA$ is used as the denominator because we need to fit *P* columns of A and rows of B.

Memory Requirements: Accordingly, as shown in Table 3.2, Solver_1 requires that two matrices (A and B) be concurrently stored in host RAM and one matrix (C) be stored in at most 96% of GPGPU device memory along with one or more blocks of A and B, the size of which are determined by the computation of *P* described above.

Data Transfers: The data transfers initiated by AHPCRCMM Solver_1 are of two types:

1. **Host-to-GPGPU block transfers**: Solver_1 iteratively (if more than one block of matrix A or B exist) transfers a block of A and a block of B from host RAM to GPGPU device memory to allow the GPGPU to produce a partial result, which is added to matrix C in GPGPU device memory. In this case, a block of A (B) is a submatrix of matrix A (B), having *P* columns (rows). The product of these two blocks is a partial result with same size as C ($RA\ x\ RA$). The number of generated partial results and, thus, the number of transfers of this type is equal to the number of blocks that comprise matrix B, which is equal to the number of blocks that comprise matrix A. Each block is only transferred from host memory to GPGPU memory once.
2. **Matrix C transfer**: After all the required partial results are computed and added to matrix C, Solver_1 transfers C from GPGPU memory to host memory.

This shows that Solver_1 has efficient data movement between the CPU and GPGPU; it transfers a total of two matrices from host memory to GPGPU memory and a total of one matrix from GPGPU memory to host memory.

Employed Performance Techniques: As shown in Table 3.2 and described in Section 3.4.1, to enhance performance, Solver_1 employs Just-in-Time (JIT) data creation whether using paged or pinned memory and JIT data deletion for only paged memory. In addition, for Class 2 experiments, AHPCRCMM_Pinned Solver_1 concurrently transfers data while executing the SGEMM kernel, which is further described in Section 3.3.2.

3.1.2 AHPCRCMM Solver_2

As mentioned above, AHPRCMM Solver_2 is used when no single matrix fits in at most 96% of GPGPU device memory. Unlike Solver_1, Solver_2 uses CUDA Streams, where a stream is a queue of instructions executed in issue order by the GPGPU device [15] to perform concurrent instruction execution by two or more streams when plausible. Both implementations of Solver_2, i.e., AHPCRCMM_Paged or AHPCRCMM_Pinned, use two CUDA Streams – we experimented with up to 16 streams and saw no meaningful difference in performance. However, only the pinned version uses the two CUDA Streams to concurrently execute the SGEMM kernel and transfer data between host memory and GPGPU device memory. To avoid memory conflicts and ensure correctness, each stream needs its own container in GPGPU device memory and host memory, thus, SN containers, in this case, SN result matrices, are needed in host memory and GPGPU memory, where SN is the number of streams.

The following process is employed by Solver_2, where i is incremented from 1 to $\frac{RA}{Q}$ for each repetition of the iterative process comprised of steps 6 through 7:

1. Allocate host memory for input matrices A and B, and output matrix C and buffer matrix TC.
2. Populate A and B in host memory, and allocate GPGPU memory for blocks from each matrix.
3. Partition A, B, and C at the host using the method described below.

4. Allocate GPGPU memory to SN block(s) of A, SN block(s) of B, and one block of C with all elements initialized to zero.
5. Transfer one block of matrix B ($P\ x\ RA$ elements) from host memory to GPGPU memory, free this block of B from host memory, and initialize $i = 1$.
6. Transfer, from host memory to GPGPU memory, the i^{th} block of dimension ($Q\ x\ P$) in the column of the submatrix of A corresponding to the *P* rows of matrix B transferred in step 3; free this block of A from host memory; and increment *i*.
7. Multiply the $Q\ x\ P$ block of A and the $P\ x\ RA$ block of B using the SGEMM kernel and store the partial results in the block of C in GPGPU memory.
8. Repeat steps 6-7 until all blocks of A in the first *P* columns of the matrix are processed.
9. Transfer the resultant block of C from GPGPU memory to host memory and then add these partial results to the associated rows of matrix C in host memory.
10. Repeat steps 5-9 for all $\frac{RA}{P}$ blocks of B.
11. Free host and GPGPU memory allocated to A, B, C and buffer matrix TC.

Employing this process, for each block (*P* rows) of B transferred to GPGPU memory, a sequence of $\frac{RA}{Q}$ blocks of A, which comprise a column of blocks of the matrix A, are transferred and multiplied with the block of B. Partial results are aggregated during this process, producing the final results matrix C.

Matrix Partitioning: Since Solver_2 assumes that the capacity of GPGPU memory does not allow matrices A, B, or C to be stored in their entirety in GPGPU memory, it must determine both *P* and *Q*, which determine the sizes of the submatrices of A, B, and C used to compute and store partial results. In this case, as shown in Figure 3.1, a block of A will be a $Q\ x\ P$ submatrix; B will be a $P\ x\ RA$ submatrix, i.e., it will consist of *P* rows of B; and C will be a $Q\ x\ RA$ submatrix, i.e., it will consist of *Q* rows of C. To determine *P* and *Q*, we first determine the largest number of rows of B that can fit in

GPGPU memory, along with a row of C and an element of A. Then, we determine the sizes of the submatrices of A and C that can be stored in the remaining GPGPU memory. This process is described next.

1. Compute *EGMS* (the Effective Global Memory Space), the total number of single-precision floating-point elements that fit in GPGPU device memory.
2. Compute $MaxP = \frac{EGMS-(SN\ x\ RA)}{1+RA}$, the maximum number of rows of B in the $P\ x\ RA$ submatrix and, thus, the largest submatrix of B that fits in GPGPU memory after allocating storage for a submatrix of C and one element of A. Given that the submatrix of C is at least one row, then $SN\ x\ RA$ in the numerator represents the minimum memory allocation required for a submatrix of C, i.e., *RA* elements. *RA* is multiplied by the number of CUDA Streams employed, in this case two, because each stream requires its own result container (to avoid memory conflicts). The denominator, $1 + RA$, represents the smallest memory allocation for submatrices A and B, i.e., one element of A and one row of B. Note that $MaxP < RA$ since otherwise matrix B would fit in GPGPU memory and Solver_1 would be used to multiply the matrices.
3. Compute $BBlocks = Ceiling(\frac{RA}{MaxP})$, the number of submatrices that comprise matrix B, which has a total of *RA* rows. BBlocks indicates how many submatrices must be transferred and processed if the largest size block is employed.
4. Compute $P = \frac{RA}{BBlocks}$. Given that BBlocks will be allocated to a block of B, which is a $P\ x\ Q$ submatrix, where $Q = RA$, then $RA = BBlocks\ x\ P$, and $P = \frac{RA}{BBlocks}$.
5. Compute $Q = \frac{EGSM-(P\ x\ RA)}{P+(SN\ x\ RA)}$, the number of rows in a block of A ($Q\ x\ P$) and the number of rows in a block of C ($Q\ x\ RA$). Based on the space remaining in GPGPU memory after allocating a block of B (a $P\ x\ RA$ submatrix), which is comprised of *P* rows of *B*, *Q* is computed by dividing the number of elements that can still be stored in GPGPU memory by the number of elements in

a row of C multiplied by 2 (*SN*) plus *P* elements of a row of A, i.e., the number of columns (P) in a submatrix of A. Q tells us how many rows of C comprise a block of C and the number of rows in a submatrix of A.

Solver_1 and Solver_2 partition matrix B in a similar way: A block of matrix B is a $P\ x\ RA$ submatrix, i.e., it is comprised of *P* rows of B. Similarly, for both solvers, a block of matrix C is a $Q\ x\ RA$ submatrix, i.e., it is comprised of Q rows of C, where $Q = RA$ in the case of Solver_1. In contrast, the two solvers partition matrix A differently. Solver_1 partitions A into $RA\ x\ P$ submatrices, while Solver_2 partitions it into $Q\ x\ P$ submatrices, where $Q \leq P < RA$. The latter inequality is true because (1) blocks of B and C have the same number of columns (RA); (2) $P < RA$ because otherwise all of B would fit in GPGPU memory and Solver_1 would be employed; and (3) a block of C has Q rows and clearly the number of rows in a block of C is less than or equal to the number of rows in a block of B, i.e., *P* (otherwise we could not have solved for *MaxP* in the way that we did, i.e., assuming that the entire GPGPU memory can store two rows of C and less than all rows of the matrix B). Thus, $Q <= P$.

Data Transfers: Accordingly, during each iteration of Solver_2, for every block of matrix B transferred from host memory to GPGPU device memory by Solver_2, $\left\lceil \frac{RA}{Q} \right\rceil$ blocks of matrix A are transferred as well. Moreover, for each block of matrix A transferred, a block of matrix C (partial results) is generated and transferred from GPGPU memory to host memory. All the blocks of matrices A and B are transferred from host memory to GPGPU memory only once. In contrast, all of matrix C ($\left\lceil \frac{RA}{Q} \right\rceil$ blocks of partial results) is transferred from GPGPU memory to host memory for every $\left\lceil \frac{RA}{Q} \right\rceil$ blocks (column of blocks) of matrix A transferred from host memory to GPGPU memory. In comparison, Solver_1 moved data associated with A, B, and C only once between host and GPGPU memory. Solver_2 matches Solver_1's efficiency w.r.t. matrices A and B, i.e., for host-to-device memory

transfers, however, its host device-to-host memory transfers could be improved. This potential improvement is left for future work.

Employed Performance Techniques: As described above, Solver_2 employs two CUDA Streams. In addition, in the AHPCRCMM_Pinned Solver_2 execution mode, it concurrently transfers data while executing the SGEMM kernel, which is further described in Section 3.3.2. Moreover, as shown in Table 3.2, it is the only solver, whether used with AHPCRCMM_Paged or AHPCRCMM_Pinned, that employs work-sharing by the host CPU and GPGPU, described in Section 3.3.3, to enhance performance.

3.3 AHPCRCMM Performance and Power Techniques

In this section we describe the power- and performance-oriented techniques used by AHPCRCMM, both the paged and pinned versions. Not included here is CUDA Streams, which is used only by Solver_2 and was described above. We indicate the techniques utilized by each solver and whether a technique is used for the paged and/or pinned versions.

3.3.1 Just-In-Time (JIT) Data Creation

Any SGEMM implementation is comprised of three phases: initialization, compute, and clean-up. For both CUBLAS and CUSUMMA, during initialization matrices A, B, and C are sequentially created and populated on the host. Next, during the compute phase (executed on the GPGPU), all matrices are kept in host memory. Finally, during clean-up (after the result has been transferred from the GPGPU to host memory), matrix C along with input matrices A and B are freed from host memory. This demands having enough host memory capacity to fit three matrices in host memory at the same time. In contrast, both the paged and pinned versions of AHPCRCMM Solver_1 employ Just-In-Time (JIT) data creation, which is a software technique that is used to reduce the memory required by a program, in this case, AHPCRCMM Solver_1. Using JIT data creation, AHPCRCMM creates and frees

data containers just before and just after they are used. Furthermore, because AHPCRCMM makes use of a co-processor (GPGPU), it overlaps the creation and/or deletion of data container(s) with co-processor instruction execution.

During initialization, AHPCRCMM Solver_1 (paged and pinned versions) sequentially creates and initializes only input matrices A and B on the host. Next, during the compute phase, just after a block from matrix B and/or one from matrix A are transferred from the host memory to GPGPU memory, they are freed from host memory. When all blocks from matrices A and B are consumed by the GPGPU and freed from host memory, output matrix C is created in host memory, and the result is transferred from the GPGPU device memory to host memory. Finally, during the clean-up phase (after the result is transferred from GPGPU memory to host memory), output matrix C is freed from host memory. The pinned version of the AHPCRCMM routine cannot take full advantage of this optimization technique; this is explained in Section 3.3.2.

3.3.2 Concurrent GPGPU Data Transfer and Instruction Execution

As mentioned before, AHPCRCMM is implemented in two different ways, i.e., with paged and pinned host memory. Paged memory allocation (e.g., allocation done by calling `malloc()`) provides main memory (RAM) data storage but the stored data has the potential of being moved from main memory to disk by the operating system (OS) if necessary. On the other hand, pinned (also known as paged-lock) memory allocation provides main memory data storage but guarantees that the stored data will remain in physical memory (RAM) and, thus, will not be evicted to disk by the OS [15]. Because a pinned section of host memory cannot be evicted, it is associated with a physical host RAM address, allowing the GPGPU to use remote direct memory access (RDMA) to copy the stored data between the host and GPGPU device memory without CPU intervention.

Moreover, the Tesla C2075 GPGPU used in this study has the capability to execute a CUDA kernel while data is being transferred between the host and device memory. Thus, we employ this

technique in Solver_1 and Solver_2 in both Class 2 and Class 3 experiments. The performance benefits of concurrent GPGPU data transfer and instruction execution are noticeable in the experiments that use AHPCRCMM_Pinned Solver_2. This is because Solver_2 is employed when none of the matrices fit in at most 96% of GPGPU memory, which results in more data transfers to perform matrix multiplication. The use of RDMA and concurrent instruction execution allows AHPCRCMM_Pinned Solver_2 to effectively hide most of the data transfers between host and device memory, with the exception of the first and last data transfers. However, one of the drawbacks of using pinned memory for concurrent host-to-device data transfer and device kernel execution is the inability to know if a block of memory has been used without introducing overhead – e.g., the CPU checking to see if the data in pinned memory can be overwritten – that eliminates the benefit of using this kind of memory allocation in the first place. This compromises the performance of AHPCRCMM_Pinned Solver_2; it cannot take full advantage of the JIT data allocation (explained in Section 3.3.1).

3.3.3 Work-Sharing between Host CPU and GPGPU

Work-sharing between the host CPU and GPGPU is used by Solver_2 only, whether the AHPCRCMM implementation uses paged or pinned memory allocation. As explained in Section 3.3.2, when using AHPCRCMM Solver_2, for every block of matrix A transferred from host memory to GPGPU device memory after GPGPU SGEMM kernel execution, one block of matrix C (partial result) is returned to a temporary container in host memory (size of blocks A and B, and partial result explained in Section 3.3.2). To add the partial results to matrix C, a POSIX thread (Pthread) is created to execute on the host. While the Pthread is performing this addition, concurrently the master Pthread executing on the host transfers a block of matrix B and/or a block of matrix A from host memory to GPGPU device memory. After the transfer completes, the same thread frees these matrix blocks from host memory. This technique is used to balance the load between the host CPU and GPGPU, and take advantage of an available, idle compute resource. In contrast to CUSUMMA's average CPU power consumption,

AHPCRCMM_Paged Solver_2 only had an increase in power consumption of 3W (see Appendix C.1.1) for the largest problem size executed on Test-bed 1 (these results are presented in Chapter 5). On the other hand, AHPCRCMM_Pinned Solver_2 had an increase of almost double. The cause of this phenomenon, which is shown in Chapter 5, is currently unknown and is left for future investigation.

3.4 COMPARISON OF CUBLAS, CUSUMMA, AND AHPCRCMM

In this section, we discuss the differences among CUBLAS, CUSUMMA, and AHPCRCMM associated with matrix partitioning and data transfers between host memory and GPGPU memory.

3.4.1. CUBLAS vs. CUSUMMA and AHPCRMM

Because CUBLAS requires input matrices A and B, and output matrix C to fit in GPGPU device memory, no partitioning of matrices at host memory is needed. Similarly, when all matrices fit in GPGPU memory (Condition 1 for AHPCRCMM Solver_1) neither CUSUMMA nor AHPCRCMM Solver_1 do partitioning of any matrix at host memory either. Thus, when all matrices fit in GPGPU memory (Class 1 experiments) there is no difference between these routines either w.r.t. partitioning or data transmission between the host and device. And, as mentioned before, they all use the SGEMM kernel to multiply matrices A and B. The only difference, with an associated performance advantage (shown in Chapter 5) of AHCRCMM versus CUBLAS is the use of the JIT data creation/deletion technique (described in Section 3.3.1).

3.4.2. AHPCRMM vs. CUSUMMA

The partitioning of matrices employed by AHPCRCMM and CUSUMMA is related to how the matrices are processed and what data must be transferred between the host and device. AHPCRCMM employs the outer-product method of matrix-matrix multiplication, while CUSUMMA uses the outer-product method only for matrices of a certain size, e.g., the two smallest matrices used in Class 2

experiments. It uses the inner-product method for larger matrices, i.e., the larger of the four Class 2 matrices and all Class 3 matrices.

As a result, as explained in Section 3.2.1, AHPCRCMM Solver_1 is most efficient in terms of host-to-device data transfers. It transfers the minimum data required to compute correct results, i.e., the size of matrices A, B, and C. In comparison, CUSUMMA transfers the same amount of data for Class 1 matrices and the two smallest Class 2 matrices.

However, for the two largest Class 2 matrices and all Class 3 matrices, CUSUMMA transfers more data than AHPCRCMM. As pictured in Figure 3.2, for the two largest Class 2 matrices, 33,792 *x* 33,792 and 35,840 *x* 35,840, CUSUMMA transfers 4.26GB and 4.78GB more data, respectively, between host and device. In the same figure, the next four sets of bars show the results for the Class 3 matrices and the largest matrix that CUSUMMA can solve for (i.e., 59,008 *x* 59,008). These experiments, which employ AHPRCMM Solver_2, show that in each scenario CUSUMMA transfers more total data between host and device than does Solver_2. The differences range from 1.78GB for the smallest matrix (i.e., 37,888 *x* 37,888) up to 25.92GB for the largest matrix that CUSUMMA can solve for.

The use of the inner-product method by CUSUMMA for the two smallest Class 2 matrices accounts for the differences in these experiments. This approach keeps a block of matrix C in GPGPU memory until it is completely solved. This requires summing the partial results of a row of blocks of matrix A and all blocks of matrix B. Accordingly, it requires that all blocks of matrix B be transferred from host to device every time a new row of blocks of matrix A are transferred. In contrast, AHPCRCMM Solver_1, using the outer-product method, transfers all blocks of input matrices A and B are once from host to device.

Figure 3.2 shows the amount of data transferred by AHPCRMM (regardless of whether paged or pinned memory allocation is employed) as well as the minimum amount of data that must be transferred

to result in correct results. As can be seen, the amount of data transferred by AHPCRCMM Solver_1 in the first two scenarios is optimal. In contrast, in the other scenarios, AHPCRCMM transfers more data than is optimal – the amount of matrix A and matrix B data transferred is optimal, however, blocks of matrix C are transferred more than once. Nonetheless, as shown in Figure 3.A, AHPCRCMM generates less data transfers between host and device than does CUSUMMA for matrices of sizes such that the input matrices A and B as well as the output matrix C fit in GPGPU memory. For the latter, smaller matrix sizes, experimentation confirms that the total amount of data transferred by CUBLAS, CUSUMMA, and AHPCRCMM, and CUBLAS is the same.

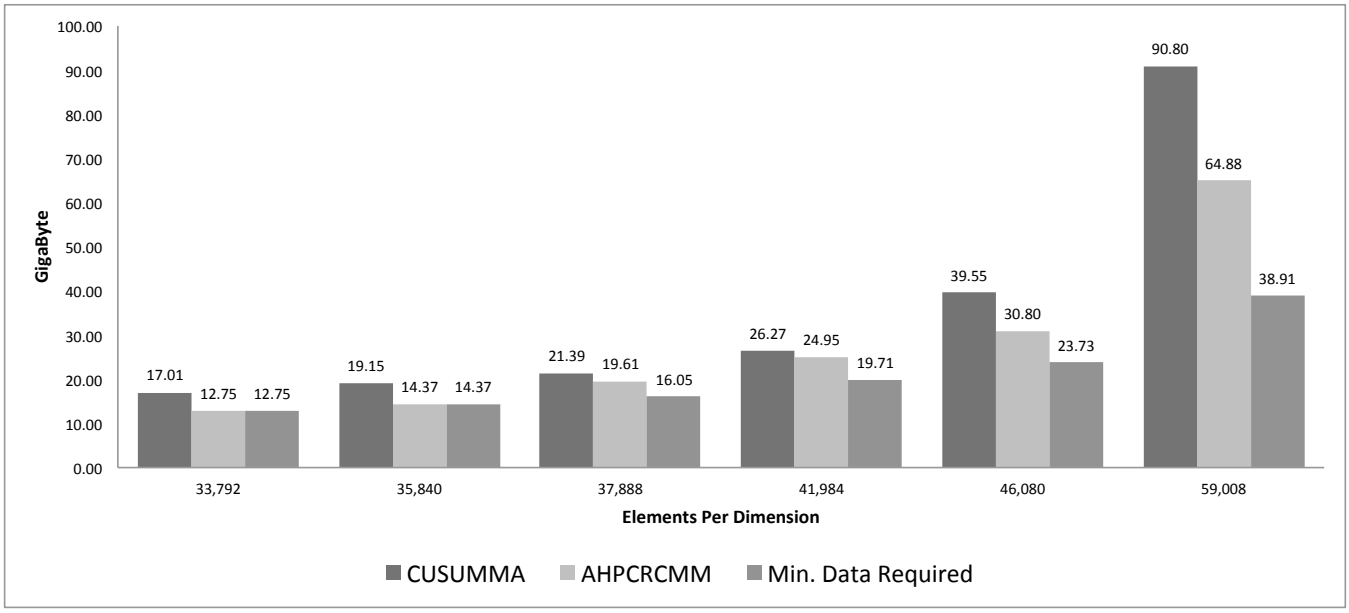

Figure 3.2: Total data transferred between host memory and GPGPU device memory

Chapter 4: Experimental Methodology

This chapter describes the methodology that we used for measuring the execution-time performance and power consumption of the selected SGEMM (single-precision, general, matrix multiplication) routines executed on two test-beds, which are described in Table 4.1. First, in Section 4.1, we discuss the basic measurements used in this research and identify the employed metrics. Then, Section 4.2 introduces and characterizes the software and hardware tools employed and Section 4.3 explains how they are used to collect measurement data. Section 4.4 delineates the sets of experiments that we conducted, justifying the selected matrix sizes used in our experiments. Finally, in Section 4.5, we describe how the raw data from the measurement tools are processed. Part of this research involved the development of a test-bed, Test-bed 1, with capabilities to measure the power consumption of the CPU, GPGPU, RAM, and total system used to execute a program. In contrast, with Test-bed 2 we can only measure the power consumed by the total system.

Table 4.1: Test-bed hardware configuration

			Test-bed	
			1	2
CPU	Name		i7-2600K	Opteron 6200
	Architecture		SandyBridge	Interlagos
	Core #		4	16
	Max. TDP		95W	115W
	GHz		3.4	2.1
	Cache	L1	64 KB	32 KB
		L2	256 KB	64 KB
		L3	8 MB	16 MB
RAM	Name		Ballistix Sport	Nanya
	Speed	DDR3	1600	1333
	Data Bandwidth (MB/sec)		12800	10600
	Voltage		1.35V	1.35V
	Size		4 x 8GB	8 x 8GB
GPGPU	Name		C2075	
	Single Precision Floating Point Performance (Peak)		1.03 Tflops	
	CUDA Cores #		448	
	Memory	Size	5.2	
		Speed	1.5 GHz	
		Bandwidth	144 GB/sec	
	System Interface		PCIe x16 Gen2	
	Power Consumption		225W	
Motherboard	Name		P8Z68-V PRO/Gen3	SuperMicro H8DGI-F
Base Software	CUDA Compiler		V0.2.1221	
	Compiler		gcc 4.6.3	
	Operative System		Ubuntu 12.04.3 LTS	
Components Measured For Power			CPU	Total System
			GPGPU	
			RAM	
			Total System	

4.1 MEASUREMENTS AND METRICS

In this research, an experiment is the execution of a matrix-matrix multiplication routine that is compiled by the nvcc compiler, provided specific input, and is executed on a specific test-bed. The execution-time performance of the routine or any part of the routine is measured by the time difference between the start and end of its execution on the test-bed.

For experiments conducted using Test-bed 1 we also report four independent measurements of power consumption, i.e., the power consumed by the CPU, GPGPU, RAM, and total system. The measurement process for each must be calibrated to start concurrently just before the routine under study begins execution. For experiments conducted using Test-bed 2, we only report, in addition to execution-time data, the power consumed by the total system. As described later in the chapter, each measurement uses a unique combination of software and hardware tools, which need to be started in a specific order.

To minimize errors and/or outliers in measured data, we executed each experiment three times and report the average of each measurement with two decimal points of precision. The reported execution time, power consumption, and energy consumption metrics are described next.

Power Consumption: In this research we use *watts* (*W*) as the power consumption metric. A watt is a unit of power that is defined as one joule (energy) per second (time); it denotes the rate at which energy is consumed. For most power measurement tools the default output values are in watts, with the exception being the RAM power measurement tool, which derives wattage from independent measurements of RAM voltage and current. The value of current is inferred from measuring the voltage difference across a current-sensing resistor embedded in a riser card that is installed between the RAM and the motherboard (more details are provided in Section 4.2.2.3).

Execution-Time Performance: Every routine under study reports execution time as a standard output expressed in seconds with two decimal points of precision. Furthermore, the execution time of a

routine is reported for three phases of execution: initialization, compute, and clean-up, as well as the total execution.

Energy Consumption: The energy measurements reported in this research are derived from the power and execution-time measurements. The average energy consumed by a routine is reported in joules (*J*) by multiplying the average wattage (*W*) consumed by the routine by its execution time (*s*), i.e.,

$$J = W \; x \; s$$

4.2 EXPERIMENTAL SETUP

In this section we introduce and characterize the software and hardware tools used in this research and explain how they are used to collect measurement data.

The function `gettimeofday()` is used to measure a routine's total execution time, as well as the execution time of its phases (initialization, computation, and clean-up). It is part of the standard library of the C programming language. `gettimeofday()` returns the current time with microsecond granularity from the system clock. To measure a code segment, e.g., an entire routine or a phase of the routine, the researcher annotates the code segment of interest by inserting in the code a call to `gettimeofday()` before and after the segment. The execution time of the code segment is the difference between the two values reported by the function. `gettimeofday()` outputs to the console, thus, the researcher must copy this data into a spreadsheet that stores execution time data and compute the execution time accordingly.

The general methodologies used to measure the power consumption of the CPU, GPGPU, RAM, and total system are described below. In the following subsections, the software and hardware tools that were employed to capture these measurements, which are identified in Table 4.2, are discussed in detail.

- CPU Power Consumption Measurement with PAPI/RAPL: The Performance API (PAPI) with the Running Average Power Limit (RAPL) component enabled was used to measure the power

consumption of the whole CPU package, which includes the memory controller embedded in the Sandy Bridge CPU installed in Test-bed 1 [19][24].

- GPGPU Power Consumption Measurement with NVIDIA-SMI: GPGPU power consumption was measured using the NVIDIA System Management Interface (NVIDIA-SMI), which is a tool built on top of the NVIDIA Management Library (NVML) [15]. NVML lets a user query GPGPU device state; one of the returned states is the current power consumption of the device.
- RAM Power Consumption Measurement: RAM power consumption was obtained from an off-line type of measurement: two riser cards were installed between Test-bed 1's motherboard and its RAM DIMMs. This test-bed's motherboard has two memory channels and two DIMMs per channel. Each riser card probes a unique memory channel and is connected to a National Instrument (NI) Data Acquisition device, which interfaces with a separate computer that runs NI SignalExpress (NISE) software. The NISE software derives the wattage of a RAM DIMM from the acquired source voltage and the voltage difference across a resistor, measured from the riser card's circuitry. An illustration of the power measurement setup of Test-bed 1 is depicted in Figure 4.1.
- Total System Power Consumption Measurement: Total system power consumption was acquired from an off-line type of measurement that uses a WT210 Yokogawa power meter, which measures total system power consumption.

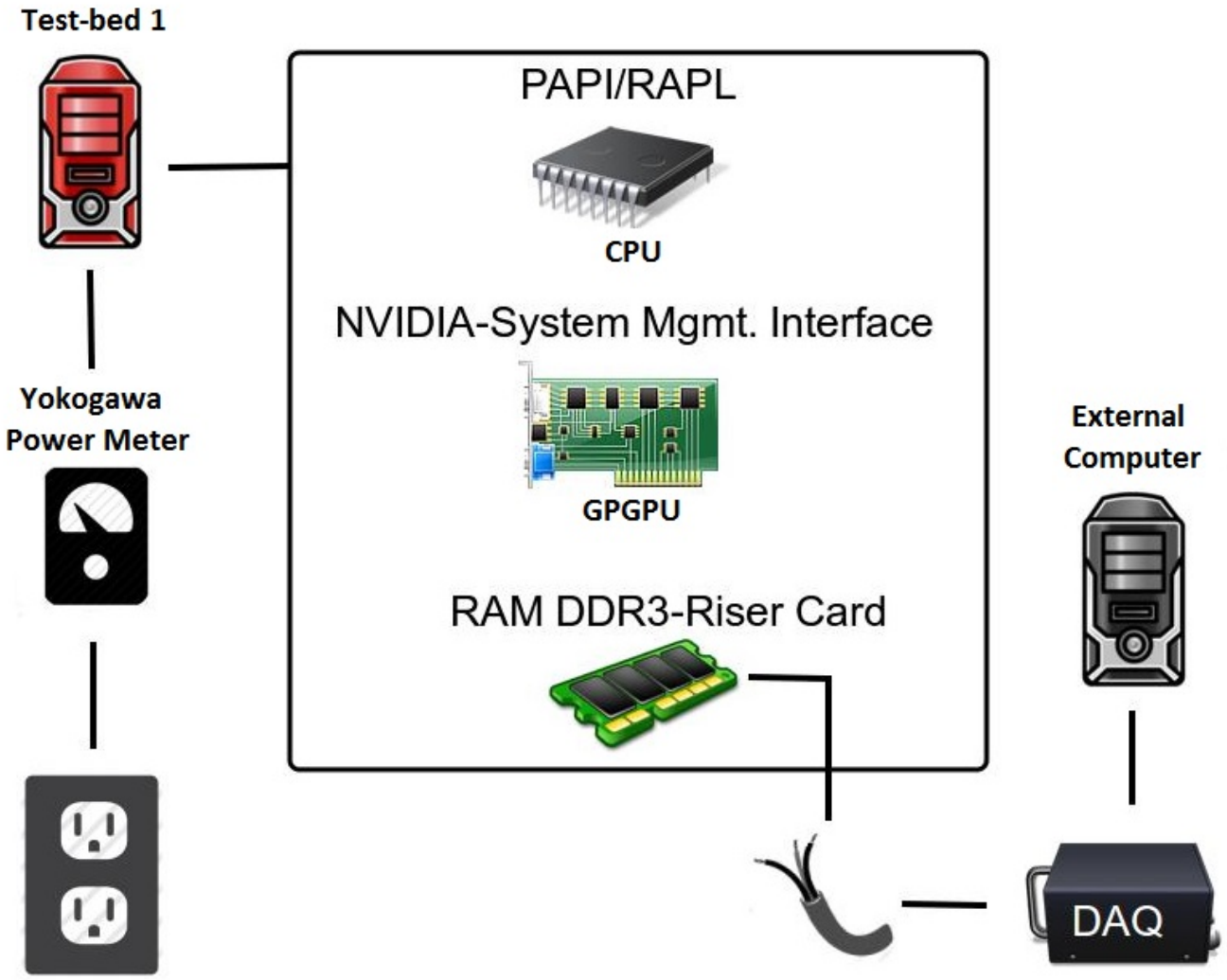

Figure 4.1: Test-bed 1 setup

Table 4.2: Measurement tools

Tool Name	Version	Type	Measured Component
PAPI/RAPL	5.1.0.2	Software	CPU
NVIDIA-SMI	4.304.88	Software	GPU
NI LabView Signal Processing	4.0.0	Software	RAM
WTViewer	8.15	Software	Total system
NI USB Digital Acquisition tool (DAQ)	USB-6216	Hardware	RAM
Yokogawa WT210	760401	Hardware	Total system
RAM DDR3 riser card	DDR3-L	Hardware	RAM

4.2.1 Software Tools

The software tools employed in this research are identified in Table 4.2 and described in this section.

4.2.1.1 PAPI/RAPL

The Performance API (PAPI) makes use of Intel's Running Average Power Limit (RAPL), which is an on-chip unit that estimates current energy usage based on a model driven by hardware counters, temperature, and leakage models [24]. To access this information, PAPI's RAPL component reads values from the Model Specific Registers (MSRs), which can be sampled in real time with millisecond resolution. In order to read these values, each core's registers must be readable by PAPI. The latter is accomplished by loading the MSR kernel module, then making each core's MSR file accessible to PAPI. The MSR kernel module is loaded by executing the following Linux terminal directive with super-user powers: **modprobe msr**. Next, we enable reading the first processor core (core zero) by issuing the following terminal directives with super-user powers: **cd /dev/cpu/0; chmod a=rw**

msr. After this step, we execute PAPI with RAPL enabled to collect power data from the CPU package. This document does not explain how to install PAPI with RAPL enabled; a detailed guide can be found at PAPI's wiki page at http://icl.cs.utk.edu/projects/papi/wiki/Main_Page.

4.2.1.2 NVIDIA-SMI

The NVIDIA System Management Interface (NVIDIA-SMI) is a tool built on top of the NVIDIA Management Library (NVML), which is a C-based API that can be used to manage and expose various GPGPU device states. The NVML library is included in the default NVIDIA display driver installation, and it does not require additional installation steps [15]. More specifically, NVIDIA-SMI is a command line utility that is meant to aid system administrators in the monitoring and management of GPGPU devices. This tool is capable of querying a GPGPU device for its current power consumption state, within a one second time window [15].

4.2.1.3 National Instruments LabView SignalExpress

National Instruments (NI) LabView SignalExpress (SE), i.e., *NISE*, is data-logging software that acquires, analyzes, and presents data captured by a data acquisition (DAQ) device. This software is paired with an NI DAQ to log and process data related to RAM DIMM power consumption (a detailed explanation appears in the next section). NISE was configured to calculate the current and power consumption of RAM DIMMs at 100-millisecond time frequency, and to save the generated data into a file. This process is started and ended manually by the researcher.

4.2.1.4 WTViewer

WTViewer is data-logging software that lets the researcher query, at a 100-millisecond time frequency, power-related data measured with a WT210 Yokogawa power meter (details regarding the WT210 power meter appear in the next section). Moreover, this software can display the voltage, current, and power used by the entire test-bed. From the WTViewer interface, the user can send a

directive to the WT210 power meter to start and end the measurement process; data is automatically saved in a file.

4.2.2 Hardware Tools

The following hardware tools, which are identified in Table 4.2, were also employed in this research.

4.2.2.1 National Instruments USB Digital Acquisition Tool

The National Instruments USB Digital Acquisition (NI DAQ) tool is a bus-powered USB module designed to deliver fine accuracy and quick sampling rates. It has 16 analog inputs with the capability of 400 KS/s (kilo-samples per second); this research only requires 10 samples per second. This component is used to acquire voltage samples from RAM (more details are provided below in the description of the RAM DDR3-Riser card) and translate them to digital signals.

4.2.2.2 Yokogawa WT210

This power meter is used to measure total system power with +- 0.1% precision and 10-millisecond measurement granularity. In order to obtain the power consumption used by the entire test-bed, the main power line of the system was sliced in half and connected through the power meter as pictured in Figure 4.1. The power meter is connected via GP-IB communication to a separate computer system to allow the WTViewer software to access measured data.

4.2.2.3 RAM DDR3-Riser Card with Power Measurement Cables for DAQ

Intrusive direct-measurement was used to measure the power consumption of RAM. A Double Data Rate 3 (DDR3) 240-pin Dual Inline Memory Module (DIMM) receives its power from the motherboard's 3.3V rail. DDR3 DIMM voltage values range from 1.20V to 2.20V (depending on the memory module) with a 0.00625V interval [P8Z68_Manual]; by default our motherboard automatically adjusts the voltage to meet the DDR3 voltage requirement. A memory DIMM riser card was installed between the motherboard DIMM enclosure and the DDR3 DIMM. We used two cable probes per

memory DIMM, one to measure the voltage difference across a current-sensing resistor (details appear below), and the other to measure the source voltage of the DIMM. Each probe connects to a DAQ input channel: one channel captures the voltage difference across the mentioned current-sensing resistor embedded in the riser card and the other captures the source voltage from the riser card circuit. The value of current (I) in a DIMM under study was calculated using Ohm's Law, $I = V/R$ (i.e., current = voltage divided by resistance) by dividing the voltage difference across the resistor (V) by the resistor's impedance (R). To calculate the total DRAM power consumption, we sum the power consumption of the installed DDR3 DIMMs, which for each DIMM is computed using Equation 4.1. The power consumption of each RAM DIMM is reported as P divided by the number of installed DIMMs. For example, if two DIMMs are installed and the average power consumed by DIMM1 and DIMM2 are 2.3 and 2.1 watts, respectively, then we report the power consumption per RAM DIMM as 2.2 watts.

$$P_{(\mathrm{W})} = I_{(\mathrm{A})} \times V_{(\mathrm{V})} \qquad \text{Equation 4.1}$$

4.3 DATA COLLECTION

In this section we explain how data was collected from each software measurement tool. For simplicity, these software tools are partitioned into two classes, i.e., in-line and out-line measurement tools. An in-line measurement tool is software that is executed by the test-bed under study. An out-line measurement tool is software that is executed by a separate machine. Tools belonging to the in-line class have the potential to cause perturbation in the final performance and/or power results. Because of this, we isolated each tool belonging to the in-line class and measured its total system power consumption, and then compared it to the system's idle power consumption. For the employed in-line tools, the difference in power was less than 1%, thus, we conclude that the induced perturbation is acceptable for the purpose of this research. Below we describe how each software tool was used to collect measurement data.

4.3.1 gettimeofday()

As described earlier, the function `gettimeofday()` is used to measure total routine execution time, as well as the execution time of its phases (initialization, computation, and clean-up). This in-line tool outputs to the console, thus, the researcher must copy the values reported by each call to `gettimeofday()` into a spreadsheet that stores execution time data and compute the execution time of each phase of execution and total execution accordingly.

4.3.2 PAPI/RAPL

The PAPI installation includes by default an example wrapper application (and source code) that makes use of the RAPL component to obtain power values of a linked application – this is an in-line tool. We modified this wrapper application to point to our SGEMM routine(s); the modified wrapper application collects the power consumption of the CPU package during the execution of the experiment. When experiment execution finishes, the average power used is presented as console standard output.

4.3.3 NVIDIA System Management Interface

NVIDIA System Management Interface, NVIDIA-SMI, runs independently of an executing application and, thus, is an in-line tool that does not need to be wrapped around an application. It queries the GPGPU device for its current power state every second; it does this before the experiment executes, while it is executing, and at the termination of the experiment. In our case, just before experiment execution, we instructed NVIDIA-SMI to query the GPGPU device for its current power state and append this value to a file. When the routine under study finishes, the researcher needs to manually terminate the NVIDIA-SMI process. After the measurement, all collected data is available in the output file for future processing. The console directive used to start this measurement is the following: **./nvidia-smi -q -l 1 -d POWER,PERFORMANCE -f ~/GPUpowerRAW.txt** .

4.3.4 NI LabView SignalExpress

The NI LabView SignalExpress software records the DRAM DIMM source voltage, as well as the voltage difference across a current-sensing resistor embedded in the intrusive DRAM DIMM riser card. This out-line software tool, which executes on an external computer system, uses these voltage values, along with specifications of the current sensing resistor, to determine the amount of current and subsequently power used. This tool has the capability to query the DAQ for these values during a 100-millisecond time window. The process of recording is manually started and stopped via the LabView SignalExpress interface by the researcher and resulting data is saved in a Microsoft Excel file.

4.3.5 WTViewer

WTViewer queries the WT210 power meter with a 100-millisecond frequency. The returned values include: voltage, amperage, and power. These values are appended to a comma-separated file (CVS) while the routine is executing. To start and end a measurement the researcher manually uses the tool's application programming interface (API). This tool is considered an out-line class of measurement because it is executed by an external computer system.

4.4 EXPERIMENTS

In the context of this research, an experiment is defined as the execution of an SGEMM routine on two square matrices, each of size $N \times N$, where N is the number of elements in a row or a column of a matrix, on a specific test-bed, i.e., Test-bed 1 or Test-bed 2, which are described in Table 4.1.

For an experiment conducted on Test-bed 1, the execution time and CPU, GPU, RAM, and total power and energy consumption of the entire SGEMM routine under study as well as its three phases were measured. For an experiment conducted on Test-bed 2, only the execution time, again for the entire routine as well as its three phases, and total system power and energy consumption were measured. Note that Test-bed 2 was used to (1) demonstrate power and performance trends across platforms and (2) to test the maximum solvable problem size of CUSUMMA (which is not possible in

Test-bed 1 due to memory limitations). We present and justify the matrix sizes used in our experiments below.

4.4.1 Matrix Sizes

For convenience we use square matrices and we express a matrix size as *N*, the number of elements in one dimension (where a matrix has *N x N* elements), along with the total size of the matrix in megabytes (MB). The 12 matrix sizes used in our experiments are presented in Table 4.4. Of these, there are four that are the most important: 21,504, 35,840, 46,080, and 59,008. Each of these represents the maximum solvable square matrix size for one of the SGEMM routines under study – 21,504 *x* 21,504 for CUBLAS, 35,840 *x* 35,840 for AHPCRCMM Solver_1, 46,080 *x* 46,080 for AHPCRCMM Solver_2, and 59,008 *x* 59,008 for CUSUMMA; the nature of these limitations are explained below. Also, the first three matrix sizes are the largest used in each set of experiments and, thus, bound the experimental domain of Class 1, Class 2, and Class 3 experiments, described in Chapter 5. Additional matrix sizes (three for Class 1 experiments, three for Class 2, and two for Class 3) were selected to acquire additional data points to better understand each routine's execution-time, power, and energy performance behavior.

4.4.1.1 Matrix Size 21,504 (~1,764MB)

The use of matrices with 21,504 single-precision elements in each dimension, i.e., matrices that each consume ~1,764MB of memory, translates to the largest square matrix that NVIDIA CUBLAS (CUBLAS) can solve on our Tesla C2075 GPGPU. Since CUBLAS requires that all three matrices (A, B, and C) are loaded in GPGPU memory [15], this limitation is imposed by the GPGPU memory capacity, which is 5.2GB for this model. In this case, with each matrix consuming ~1,764MB, the GPGPU memory size required by CUBLAS is approximately 5.2GB. To multiply larger matrices on this GPGPU requires a SGEMM routine that employs host matrix tiling. Currently, CUBLAS does not support this capability but CUSUMMA and the AHPCRCMM routines do.

4.4.1.2 Matrix Size 35,840 (~4,900MB)

As explained in Chapter 3, the AHPCRCMM routine employs two matrix-multiplication solvers. The use of square matrices with 35,840 single-precision elements in each dimension, each of which consumes ~4,900MB of memory, translates to the largest matrix that Solver_1 can solve on our Tesla C2075 GPGPU. For larger matrices, Solver_2 is used. Since AHPCRCMM Solver_1 requires that the entire result matrix C fit in at most 96% of GPGPU memory (discussed in Chapter 3), this limitation is imposed by the Tesla C2075 memory capacity of 5.2GB.

4.4.1.3 Matrix Size 46,080 (~8,100MB)

The use of square matrices with 46,080 *x* 46,080 single-precision elements, each of which consumes ~8,100MB of memory, translates to the largest matrix that Solver_2 can solve when executing on Test-bed 1. For larger matrix sizes (i.e., with more than 46,080 elements per matrix) Test-bed 2 is used. Since AHPCRCMM Solver_2 requires four matrices to be resident in the host's DRAM at the beginning of execution (discussed in Chapter 3), this limitation is imposed by Test-bed 1's DRAM capacity. Larger matrix sizes were tested using Test-bed 2, which has double the DRAM capacity of Test-bed 1 (hardware specifications for the test-beds are provided in Table 4.1).

4.4.1.4 Matrix Size 59,008 x 59,008 (~13,282MB)

The use of square matrices with 59,008 elements in each dimension, each of which consumes ~13,282MB of memory, translates to the largest matrix that CUSUMMA can solve on our test-beds. CUSUMMA tiles matrices A, B, and C in blocks with size Ab, Bb, and Cb, such that $\mathrm{Ab} + \mathrm{Bb} + \mathrm{Cb} <= \mathrm{GPGPU}_{Memory}$. Up to this matrix size this condition holds true, but for larger matrices the tiling code segment of the routine erroneously assigns block sizes that exceed the GPGPU memory capacity, thus, causing an allocation error in the GPGPU device.

Table 4.4: Selected matrix sizes

Single-Precision Floating-Point Elements Per Column/Row	Matrix Size in Megabytes
9,216	324
13,312	676
17,408	1,156
21,504	1,764
25,600	2,500
29,696	3,364
33,792	4,356
35,840	4,900
37,888	5,476
41,984	6,724
46,080	8,100
59,008	13,282

4.5 DATA PROCESSING

In this section we discuss how we extract, process, and use the data generated by the measurements collected as described in Section 4.3. At creation, raw data from each measurement resides in its corresponding container and/or computer. A set of clean-up and organization steps were applied to this raw data, before the data was analyzed. The complete set of steps followed is listed below:

1. Preprocess raw data by deleting unnecessary data introduced by measurement program or apparatus by default, such as labels or special characters.
2. Label the preprocessed data file according to our naming convention: component _routine_problem size, where component can be CPU, GPGPU, RAM, Test-bed 1, or Test-bed

2; routine can be CUBLAS, CUSUMMA, AHPCRCMM_Paged, or AHPCRCMM_Pinned; and problems size can be any number (without commas) found in the first column of Table 4.4.

3. Move the data file to a unique folder located on the computer where the data analysis will be performed.
4. After all files have been moved to the analysis machine, extract data from each file and add it to a single data file. This process includes grouping related data, for example storing execution-time performance data and power data in different data sheets. All resulting data files are stored in the analysis machine, and a back-up copy is made in a cloud service.
5. Analyze data by applying appropriate function(s) to the data (e.g., average, mean, etc.).
6. Create explanatory plots from these data files.

Next, we discuss in more detail how each type of measurement data was processed. Using the software and hardware tools described previously in this chapter, we created five Microsoft Excel spreadsheets. These spreadsheets store data for each experiment conducted that quantifies: (1) execution-time performance, (2) GPGPU average power consumption, (3) CPU average power consumption, (4) RAM average power consumption, and (5) total system average power consumption. Accordingly, each spreadsheet is associated with a particular set of software and/or hardware tools, i.e., (1) `gettimeofday()`, (2) NVIDIA-SMI, (3) PAPI/RAPL, (4) NI SignalExpress, and (5) WTViewer, respectively. Each Microsoft Excel spreadsheet has an entry for every experiment.

4.5.1 Execution Time Data

As mentioned in Section 4.3.1, the function `gettimeofday()` is used to measure the total execution time of a routine, as well as the execution time of its phases (initialization, computation, and clean-up). `gettimeofday()` outputs to the console, thus, the researcher must copy this data into the spreadsheet that stores execution time performance data.

4.5.2 Average CPU Power Consumption Data

PAPI/RAPL generates only one numeric value (average CPU-package power) per experiment. The researcher copies this value from the console window directly into a formatted table in the spreadsheet that stores only CPU power consumption data. This table connects each average CPU power value with the name of the associated experiment, i.e., the studied SGEMM routine, matrix size, and test-bed. The data in this table is plotted to show the behavior of average CPU power consumption across the experiments conducted using Test-bed 1 and Test-bed 2.

4.5.3 GPU Power Consumption Data

NVIDIA-SMI generates log-like output. For every response from the GPGPU, this tool outputs various parameters (e.g., timestamp, driver version, attached GPUs, power drawn). After an experiment, the raw data file created by NVIDIA-SMI during the experiment contains values that are not related to this research and, thus, need to be excluded. Accordingly, the researcher must extract from this raw data file values representing power drawn and move them to the spreadsheet that stores all data related to GPGPU power consumption. The data in this spreadsheet is plotted to show the behavior of GPU power consumption across the experiments conducted using Test-bed 1.

4.5.4 RAM Power Consumption Data

As explained in Section 4.2.1.3, the NI LabView SignalExpress software is executed on a separate computer, which is connected to a data acquisition (DAQ) device. As shown in Figure 4.1, probe cables are connected from the test-bed's RAM memory riser card to the DAQ. This software has the capability to query the DAQ for measurements every 10 milliseconds. The data is presented to the researcher as a two-dimensional graphic image, where the X-axis is time and the Y-axis is voltage, current, or power depending on which signal is selected. Recording of these values is started and stopped by the researcher, and the values are saved in a Microsoft Excel spreadsheet. This spreadsheet contains a large volume of data and tabs, which are created by default. The researcher accesses this file

and extracts the column of data that represents the amount of power (in watts) used by each measured RAM DIMM. Extracted data is stored in the spreadsheet that stores all data related to RAM power. The data in this spreadsheet is plotted to show the behavior of RAM power consumption across the experiments conducted using Test-bed 1.

4.5.5 Total System Power Consumption Data

As described above in Section 4.2.1.4, the WTViewer software is executed on a separate computer, which has the WT210 Yokogawa power meter connected to it. The output of this program is a CSV file that stores the raw data in a column-wise manner. Each column represents each queried parameter from the WT210 power meter, i.e., current, voltage, and power consumption. The researcher extracts the data in the column that provides power consumption values and saves them in the spreadsheet that stores all total system power consumption data. The data in this spreadsheet is plotted to show the behavior of total system power consumption across the experiments conducted using Test-bed 1 and Test-bed 2.

Chapter 5: Experimental Results and Analysis

This chapter presents and analyzes the results of our experiments. The experiments provided data regarding the execution time, power consumption, and energy consumption of the three SGEMM routines under study, which are described in Chapter 3, executed on the test-beds described in Table 4.2. All experiments are run on Test-bed 1, while only a small number are executed on Test-bed 2 to further investigate experimental results. The SGEMM routines studied are CUBLAS, CUSUMMA, and our SGEMM routine, AHPCRCMM, which is implemented using paged and pinned memory and, is represented by AHPCRCMM_Paged and AHPCRCMM_Pinned, respectively. The results of the experiments are organized into classes to facilitate analysis; each class of experiments, i.e., experiments that characterize the execution of each SGEMM routine on small, medium and large matrices are presented in Sections 5.2, 5.3, and 5.4, respectively. Finally, we summarize the results in Section 5.5.

5.1 Experiment Classes

Each experiment, which is driven by CUBLAS, CUSUMMA, AHPCRCMM_Paged, or AHPCRCMM_Pinned, conducted as part of this study belongs to one of three classes:

Class 1: **Small Matrix**: Input matrices A and B as well as output matrix C fit in GPGPU memory, which for both Test-bed 1 and Test-bed 2 has a capacity of 5.2GB. CUBLAS, CUSUMMA, AHPCRCMM_Paged (Solver_1), and AHPCRCMM_Pinned (Solver_1) drive the four sets of experiments in this class, one for each of four different matrix sizes. CUBLAS requires that all three matrices fit in GPGPU memory; thus, all CUBLAS experiments belong to this class. And, for the largest matrix size in this class, i.e., the largest matrix in this study that CUBLAS can solve for, all of GPGPU memory is required to store matrices A, B, and C.

Class 2: **Medium Matrix**: At most one matrix fits in at most 96% of GPGPU memory. As a result, only CUSUMMA and the AHPCRCMM routines (Solver_1) drive the four sets of experiments in this class, one for each of four different matrix sizes. Note that the largest

size matrix in this class, i.e., the one that fits in 96% of GPGPU memory is the largest square matrix that AHPCRCMM Solver_1 can solve for.

Class 3: **Large Matrix**: The sizes of the matrices are such that none of the matrices A, B, and C fit in GPGPU memory. As a result, only CUSUMMA and the AHPCRCMM routines (Solver_2) drive the three sets of experiments in this class, one for each of three different matrix sizes. The largest of these represents an upper bound for AHPCRCMM Solver_2 – this is the largest size matrix that it can solve on Test-bed 1. This limitation is due to the fact that Solver_2 must have enough host RAM capacity to store four matrices of this size.

Thus, three (Class 3) or four (Classes 1 and 2) sets of experiments comprise each class. In each class either three or four of the SGEMM routines under study are used to multiply three or four different sized square matrices. Accordingly, execution time, power consumption, and energy consumption data are organized into experiment classes. We focus individually on the results for each class because our AHPCRCMM routines, as well as CUSUMMA, process different sized matrices with different strategies. For example, AHPCRCMM follows a different process for the three different classes of matrices, i.e., small, medium, and large, with which we experiment. In particular, it employs different techniques, which are described in Chapter 3, to decrease execution time and/or power consumption and, thus, energy consumption.

5.2 Performance Results for Class 1 (Small Matrix) Experiments

In this section we present and analyze the results of the experiments in Class 1 executed on Test-bed 1, referring to Test-bed 2 results when appropriate. Again, Class 1 experiments perform matrix multiplication on input square matrices, A and B, with output matrix C, all of which fit in GPGPU memory, which is 5.2GB for both Test-bed 1 and Test-bed 2. In particular, we experiment with 9,216 *x* 9,216 (size A), 13,312 x 13,312 (size B), 17,408 *x* 17,408 (size C), and 21,504 *x* 21,504 (size D) matrices with single-precision floating-point data. Note that our comparison of the performance of

AHPCRCMM routines to that of CUBLAS is based only on these experiments since CUBLAS requires that all three matrices fit in GPGPU memory. Sections 5.2.1, 5.2.2, and 5.2.3 present and analyze performance data for Class 1 experiments in terms of execution time, power consumption, and energy consumption, respectively, while Section 5.2.4 summarizes the results and presents an analysis.

5.2.1 Execution Time

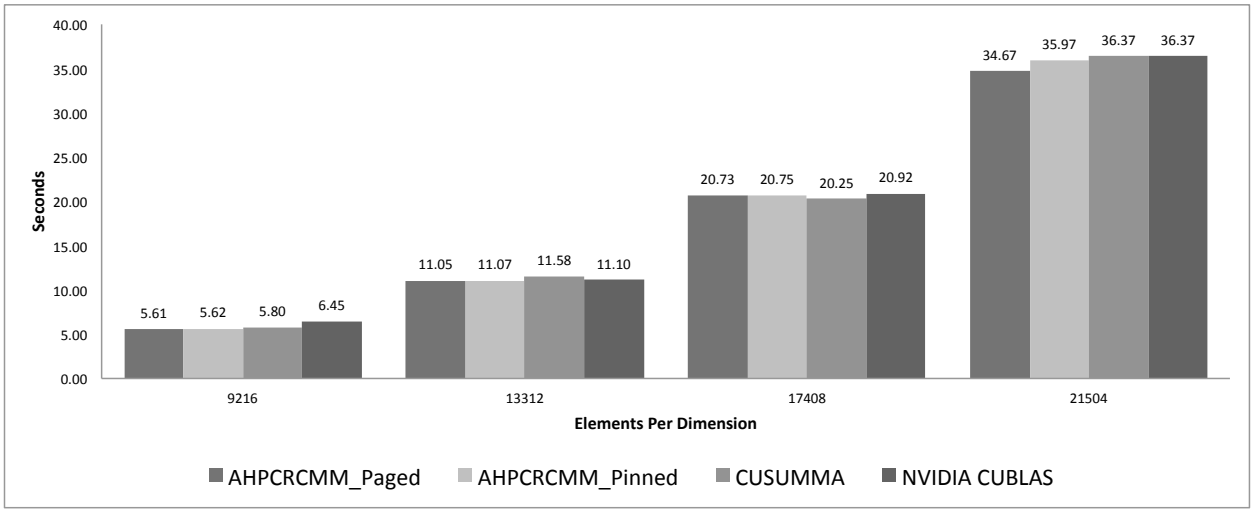

Figure 5.1.1 -- Class 1 (small matrix) experiments: total execution time

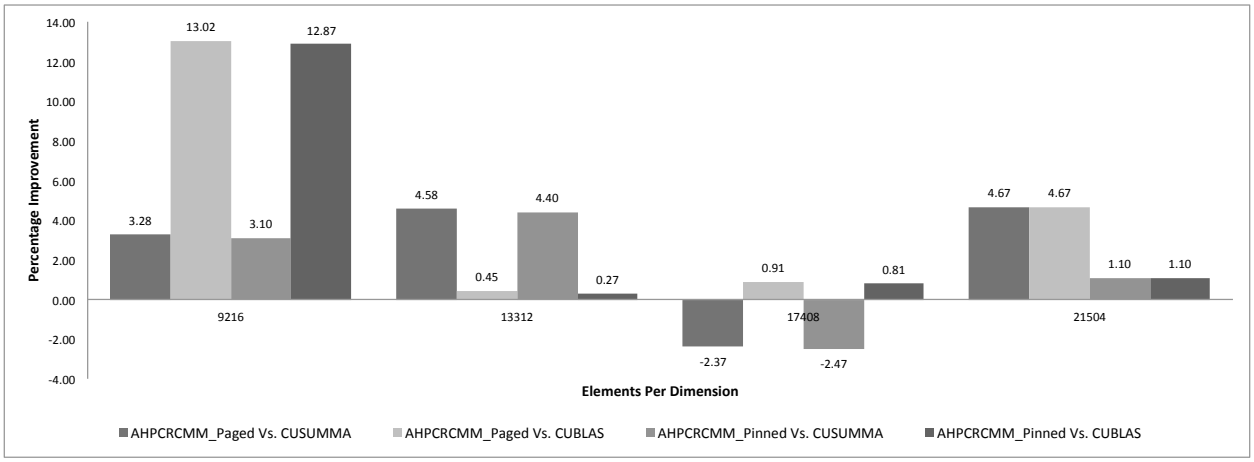

Figure 5.1.2 – Class 1 (small matrix) experiments: total execution time improvement

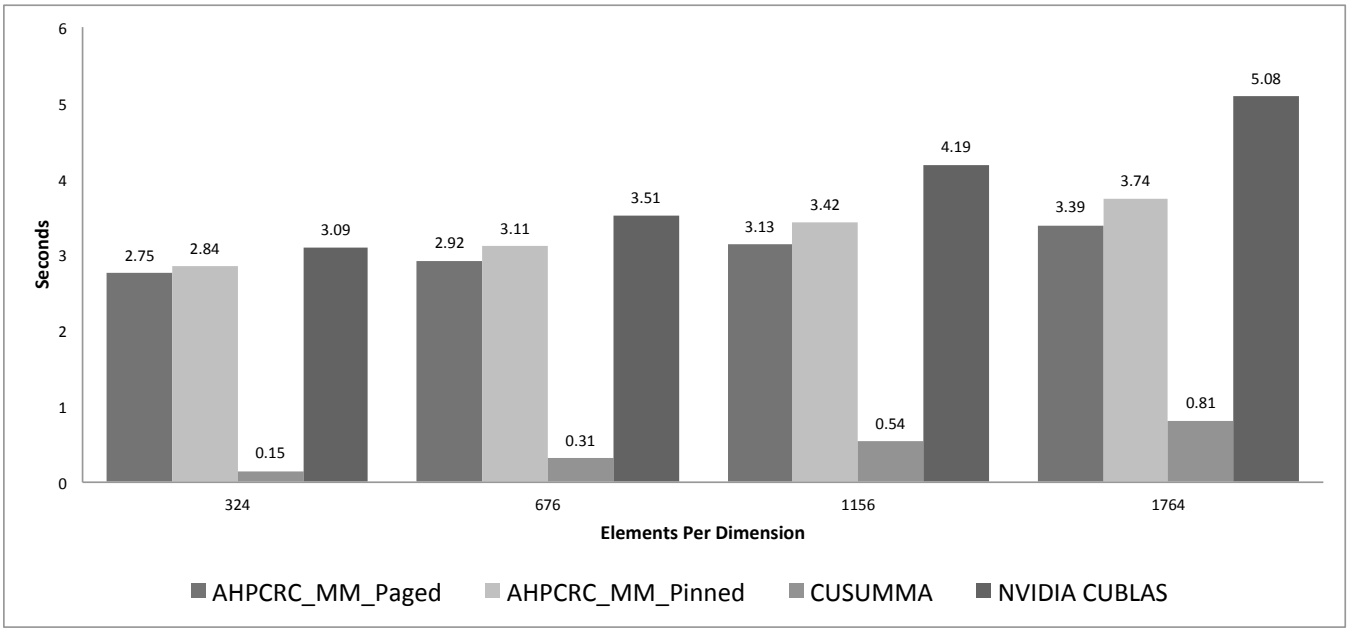

Figure 5.1.3 -- Class 1 (small matrix) experiments: initialization phase execution time

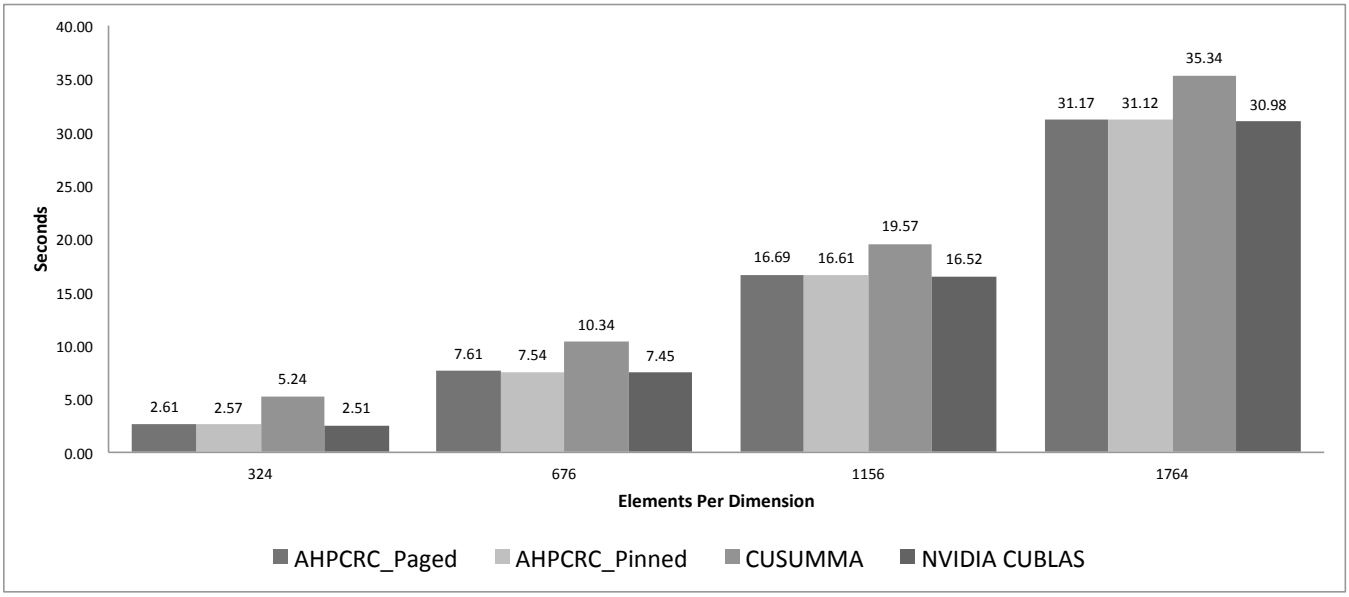

Figure 5.1.4 -- Class 1 (small matrix) experiments: compute phase execution time

Figure 5.1.1 presents the total execution time data for all Class 1 experiments, while Figure 5.1.2 presents the execution-time percentage improvement of AHPCRCMM (paged and pinned) as compared

to CUBLAS and CUSUMMA. If the percentage improvement is negative (positive), it means the AHPCRCMM routine is slower (faster).

For the Class 1 experiments, all four SGEMM routines perform matrix multiplication in a similar way, i.e., via the following six-step process:

1. Allocate in host memory input matrices A and B, and output matrix C;
2. Initialize matrices A, B, and C;
3. Transfer matrices A and B to GPGPU device memory (GPGPU memory);
4. Execute the CUDA SGEMM kernel on the GPGPU device;
5. Transfer the result matrix C from GPGPU memory to host memory; and
6. Free host and GPGPU memory regions that store input matrices A and B, and result matrix C.

Because all four routines follow this process, as shown in Figure 5.1.1, they exhibit similar execution-time performance, i.e., differences in execution times are between .27% and 13.02%, across all experiments, with only two, which involve the smallest matrix size, differ by over 5%. Additionally, as depicted in Figure 5.1.2, no single routine consistently performs better across all four sets of experiments and only two show an execution time improvement that is over 5%, i.e., AHPCRCMM_Paged and AHPCRCMM_Pinned have execution times that are about 13% faster than that of CUBLAS for the smallest matrix size, size A. In contrast, the two AHPCRCMM routines provide a savings of less than .5% execution time for size B, approximately 1% for size C, and approximately 5% (AHPCRCMM_Paged) and 1% (AHPCRCMM_Pinned) for size D.

As compared to CUSUMMA, AHPCRCMM_Paged and AHPCRCMM_Pinned have very similar performance and the differences in execution times are small, i.e., between 2.37% and 4.67%. Both have a performance advantage of less than 5% for three of the four matrix sizes, i.e., for size A, B, and D, and perform slightly worse (2.37%/2.47% less) for size C.

To understand why AHPCRCMM executes faster than CUBLAS, we examined the execution-time data for the three phases of each routine, i.e., initialization, computation, and clean-up. These phases are well defined in CUBLAS and AHPCRCMM, but are not well defined in CUSUMMA because its initialization and compute code sections are tightly coupled and cannot be isolated. Because of this, we are not able to compare the execution time of the initialization and compute phases of CUSUMMA with that of CUBLAS or AHPCRCMM.

Comparing CUBLAS and AHPCRCMM, we see in Figure 5.1.4 that the execution times of the compute phase for CUBLAS and the AHPCRCMM routines are similar – a difference of less than 2% on average, with CUBLAS performing slightly better. Thus, we observe a larger difference in execution times of the initialization phase, the execution-time performance improvement garnered by the AHPCRCMM routines is due to initialization and/or clean-up. Referring to Figure 5.1.3, which presents the execution time of the initialization phase of the four SGEMM routines, we notice that the AHPCRCMM routines perform better than CUBLAS during the initialization phase, i.e., 21% for AHPCRCMM_Paged and 16% for AHPCRCMM_Pinned on average, and that the execution time of CUBLAS' initialization phase increases more rapidly with matrix size. This performance advantage attained by AHPCRCMM_Paged is due to Just-In-Time (JIT) data creation, which is explained in more detail in Chapter 3. Note that JIT data deletion is not possible when pinned memory in the host is employed.

However, it is important to note that this advantage diminishes with increasing matrix size. This is because for all the routines as the matrix size increases, the execution time of the computation phase increases faster than does the time spent on initialization, which can be noticed by comparing Figures 5.1.3 and 5.1.4. For CUBLAS and the AHPCRCMM routines, while the execution time of the computation phase is comparable across the experiments, the execution time of the initialization phase is approximately 45% of the total execution time for multiplying the smallest matrix size used in Class 1

experiments (Size A), about 30% for Size B, 20% for Size C, and 15% for Size D. This explains the relatively large performance advantage attained by AHPCRCMM for only the smallest matrix multiplied. The performance improvement provided by JIT data creation impacts the execution time of the initialization phase, the impact of which decreases with increasing matrix size.

In summary, the data presented in this section shows that for the Class 1 experiments, because all four routines perform matrix multiplication in essentially the same way, they exhibit similar execution-time performance. As expected, for each routine execution time increases with matrix size but no one routine consistently outperforms across all four sets of experiments. However, AHPCRCMM does outperform CUBLAS in all Class 1 experiments. For one experiment – with the smallest matrix size (Size A) – both AHPCRCMM routines execute approximately 13% faster and for the other experiments they attain an improvement in execution time of approximately 4.5% for Size B, 1% for Size C, and 5% (paged) and 1% (pinned).

In comparison to CUSUMMA, for all Class 1 experiments the AHPCRCMM routines execute all matrices, except 17,408 x 17,408, faster; for the 17,408 x 17,408 matrix, they exhibit a performance disadvantage of approximately 2.5%.

The performance advantages exhibited by AHPCRCMM are largely due to JIT data creation in the initialization phase. Since the performance impact of this technique only impacts the initialization phase, its impact decreases with increasing matrix size. This is because for all the routines as the matrix size increases, the execution time of the computation phase increases faster than does the time spent on initialization. This explains the relatively large performance advantage attained by AHPCRCMM for only the smallest matrix multiplied.

5.2.2 Power Consumption

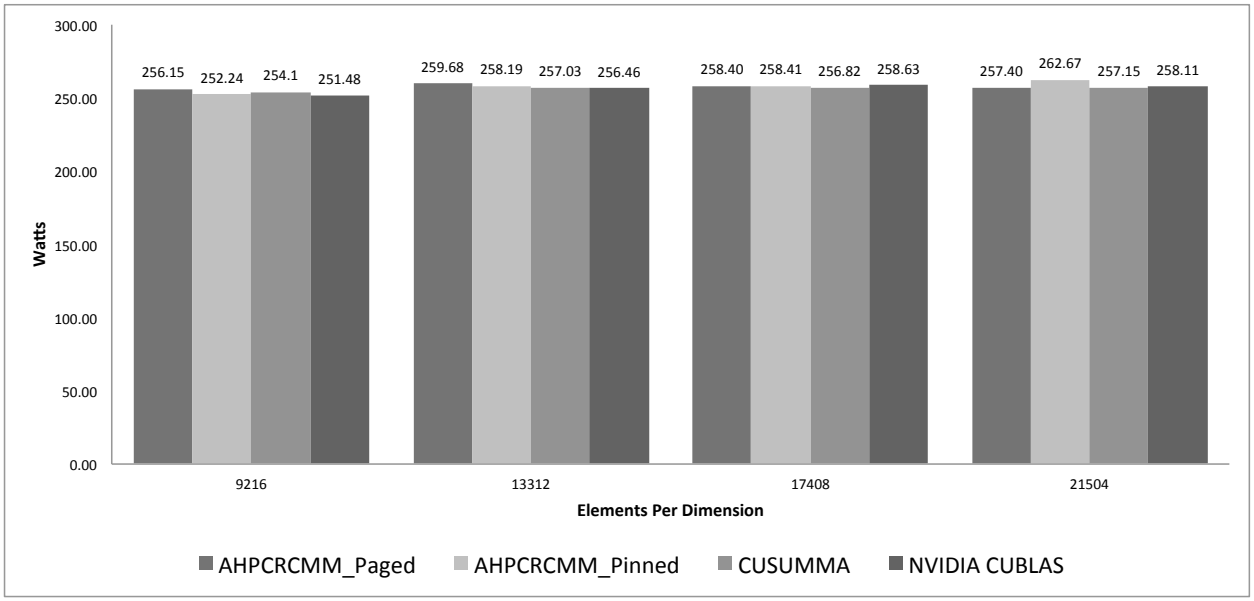

Figure 5.2.1 – Class 1 (small matrix) experiments: total average power consumption

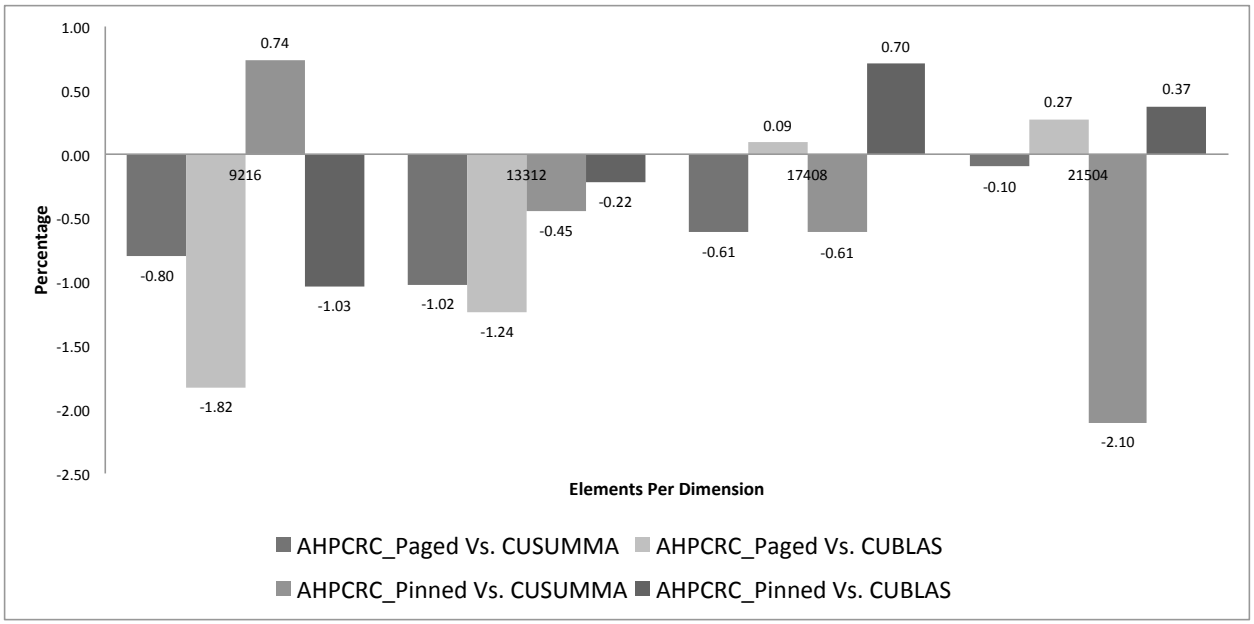

Figure 5.2.2 – Class 1 (small matrix) experiments: total average power consumption improvement

Figure 5.2.1 presents the total average power consumption for Class 1 experiments. For the same set of experiments, Figure 5.2.2 presents the average power consumption percentage improvement of

AHPCRCMM (paged and pinned) routines compared to CUBLAS and CUSUMMA. If the percentage improvement is negative (positive), it means the AHPCRCMM routine has higher/worse (lower/better) average power consumption.

Since the four SGEMM routines under study perform matrix multiply in a similar way when executing matrices of the sizes used in Class 1 experiments, as shown in Figure 5.2.1, the total average power consumption is essentially uniform across all routines with a maximum difference of less than 2.10% (see Figure 5.2.2), which is the difference between the average power consumption of AHPCRCMM_Pinned and CUSUMMA for the largest matrix size used in this class of experiments.

5.2.3 Energy Consumption

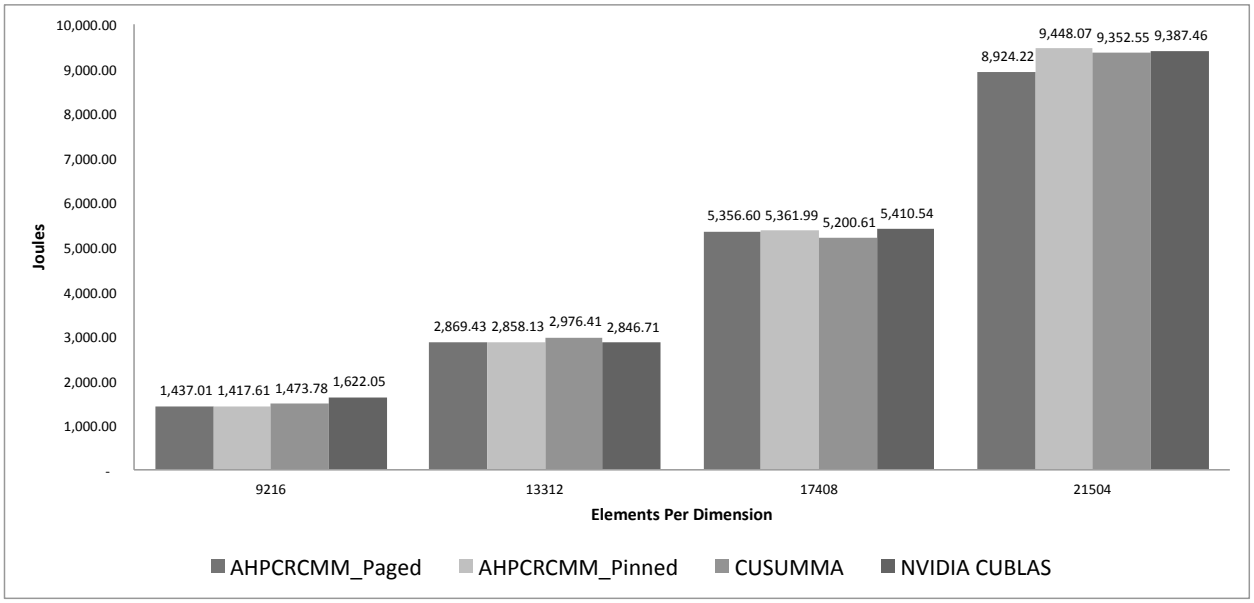

Figure 5.3.1 – Class 1 (small matrix) experiments: total average energy consumption

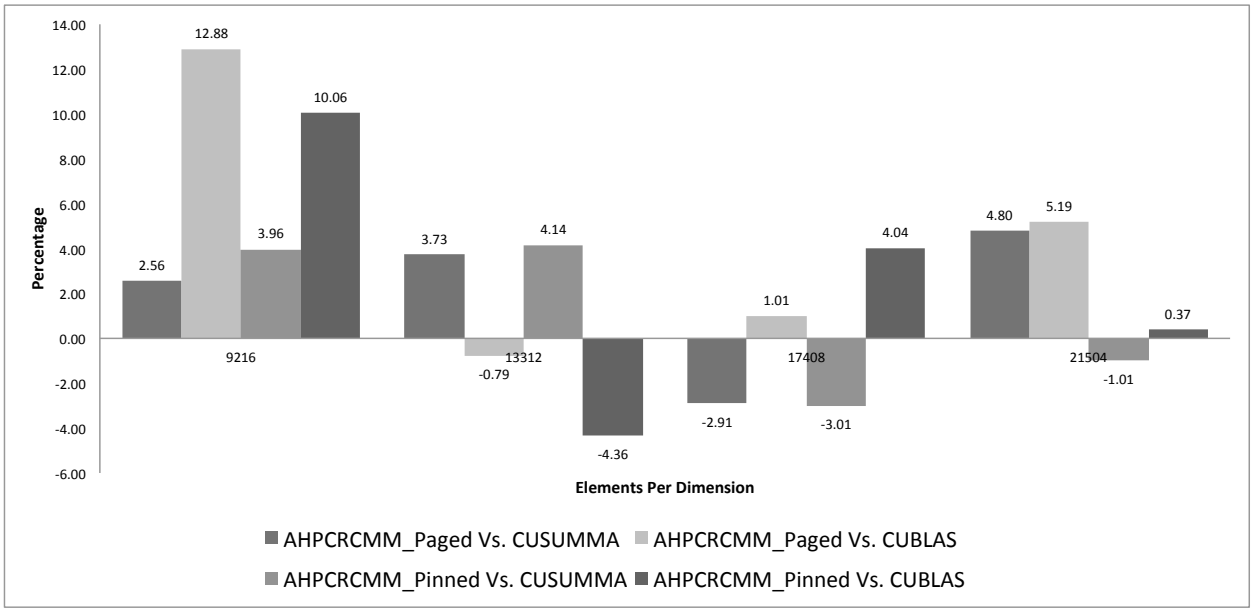

Figure 5.3.2 – Class 1 (small matrix) experiments: total average energy consumption improvement

Figure 5.3.1 presents the total average energy consumption for the four SGEMM routines when they multiply the smallest matrices used in this study, which allow the two input matrices and one output matrix to fit in the GPGPU memory of Test-bed 1, i.e., 5.2GB. For the same set of experiments, Figure 5.3.2 presents the percentage improvement of the total average energy consumption of AHPCRCMM (paged and pinned) compared to that of CUBLAS and CUSUMMA. If the percentage improvement is negative (positive), it means that the AHPCRCMM routine has higher/worse (lower/better) average energy consumption.

Since the four SGEMM routines execute matrix multiply in a similar fashion for the matrices studied in Class 1 experiments, as shown in Figure 5.3.2, the improvement in average energy consumption attained by the AHPCRCMM routines ranges from -5% to 5% with the exception of the smallest matrix size, i.e., 9,216 x 9,216, for which the total average energy consumption improvement of AHPCRCMM routines as compared to CUBLAS is between 10% and 13%. This larger improvement is due to the associated total execution-time improvement of approximately 13%, which is due to JIT data creation. As explained in Section 5.2.1, JIT data creation contributes to the lower total execution

time of the initialization phase of AHPCRCMM, but its effect decreases as the matrix size increases (As the matrix size increases, the execution times of the initialization and clean-up phases represent a decreasing percentage of execution time, while the execution time of the computation phase represents a larger percentage).

To explore how a larger RAM might affect performance, we used Test-bed 2 to multiply the largest of the Class 1 matrices – Test-bed 2 has double the RAM capacity of that of Test-bed 1, a comparable CPU, and the same GPGPU. These experiments showed that the total average energy consumption improvement of AHPCRCMM_Paged, as compared to CUBLAS (CUSUMMA), was 4.5% (7.2%), which is associated with an 8.3% (8.8%) decrease in execution time and a 4.3% (2.3%) increase in total average power consumption. The total average energy consumption improvement of AHPCRCMM__Pinned, as compared to CUBLAS (CUSUMMA), was 3.35% (5.9%), which is associated with improvements of 4.9% (5.4%) in execution time and 1.8% (.1%) in total average power consumption, these results are presented in Appendix A.

5.2.4 Summary

In comparison to CUBLAS, AHPCRCMM always performs better in terms of execution time, i.e., .27% to 13.02% better, but the performance advantage is not monotonically increasing with matrix size for either the paged or pinned versions. And, the paged version always performs slightly better than the pinned version. The biggest performance advantage is seen when multiplying the smallest matrices. Although the differences in power consumption are less than 2.11%, energy performance differences, which range from -4.36% to 12.88% improvement, do not correlate with the execution-time differences except in the case of the smallest matrix size.

Both AHPCRCMM routines also perform better than CUSUMMA in terms of execution time for all matrix sizes used in Class 1 experiments (with a 1.10% to 4.67% performance advantage), except 17,408 x 17,408, for which they exhibit a performance disadvantage of approximately 2.5%. Again, the

execution-time behavior does not translate directly to energy performance except in the case of the smallest matrix size. For the other matrix sizes, the differences in total average energy consumption range from -3.0% to 4.08%.

In summary, in the Class 1 experiments, AHPCRCMM performs better than CUBLAS in terms of execution time, with AHPCRCMM_Paged achieving slightly better performance. However, there is no clear winner in terms of power or energy performance. The execution-time performance advantage of AHPCRCMM is largest for the smallest matrix (appx. 13%) and then decreases to less than 5% for the other three matrix sizes. In terms of CUSUMMA, AHPCRCMM performs better in terms of execution time for all but one matrix size – the performance advantage is always less than 5% and the performance disadvantage is less than 2.5%.

The performance advantage of AHPCRCMM vs. CUBLAS comes from the only difference between how these different SGEMM routines multiply matrices in this size range. JIT data creation provides this performance advantage but it is only garnered in the initialization phase, where AHPCRCMM employs JIT data creation. And, since for all routines the contribution of the computation phase to total execution time increases with increasing matrix size, while the contribution of the initialization phase decreases, the performance advantage garnered by JIT data creation also decreases. Consequently, it is largest for the experiments with the smallest matrix size (around 13% in terms of execution time and 10-13% in terms of energy consumption) and decreases thereafter.

Finally, it should be noted that for the relatively small matrix sizes used in the Class 1 experiments, execution times are between 5 and 37 seconds, and a difference of around 13% in execution time (the largest performance improvement garnered by AHPCRCMM_Paged vs. CUBLAS) amounts to an improvement of only .83 seconds, the difference between 6.45 and 5.61 seconds. Analogously, the difference of around 13% in energy consumption (the largest performance

improvement garnered by AHPCRCMM_Pinned vs. CUBLAS) amounts to an improvement of 204.44 joules, the difference between 1,622.05 and 1,417.61 joules.

5.3 Performance Results For Class 2 (Medium Matrix) Experiments

In this section we present and analyze the results of the experiments in Class 2 executed on Test-bed 1, referring to Test-bed 2 results when appropriate. Again, each Class 2 experiment is driven by one of the four SGEMM routines under study to multiply square matrices, such that at most one matrix fits in at most 96% of GPGPU memory, which for our experimental platforms has a capacity of 5.2GB. Because CUBLAS requires that all three matrices fit in GPGPU memory, Class 2 and Class 3 experiments are used to compare the performance of only AHPCRCMM routines with that of CUSUMMA. Also, the largest size matrix used in these experiments is the largest size that AHPCRCMM Solver_1 can solve for; recall that Solver_1 requires that at most one matrix fits in at most 96% of GPGPU memory.

The AHPCRCMM routines perform matrix multiplication similarly for matrix sizes used in Class 2 experiments, while CUSUMMA performs matrix multiplication in a different way for these size matrices. The execution of CUSUMMA and AHPCRCMM when working on the matrix sizes used in Class 2 experiments is described in Chapters 2 and 3, respectively. The general five-step process followed by AHPCRCMM routines (paged and pinned) and CUSUMMA for the matrix sizes used in Class 2 experiments is:

1. At the host processor partition input matrices A and B, as well as output matrix C (if applicable) into blocks.
2. Transfer a block from each input matrix to GPGPU device memory.
3. Execute the CUDA SGEMM kernel on the GPGPU device.
4. Return a partial result from the GPGPU device to the host memory (if applicable).
5. Free associated host and GPGPU device memory.

Repeat steps 2 through 4 until all blocks of matrices A and B are processed.

However, when working with Class 2 matrix sizes, AHPCRCMM (paged and pinned) and CUSUMMA have the following key differences in steps 1, 2, and 4: (1) the block sizes used by each routine differ; (2) the total amount of data transferred between the host and GPGPU device differ: and (3) for the two largest matrix sizes in this class CUSUMMA transfers partial results of matrix C, whereas AHPCRCMM stores all of matrix C in GPGPU memory for all Class 2 matrices.

In Class 2 experiments we study the performance of the multiplication 25,600 (size E), 29,696 (size F), 33,792 (size G), and 35,840 (size H) single-precision floating-point matrices. Sections 5.3.1, 5.3.2, and 5.3.3 present data that is used to analyze performance in terms of execution time, power consumption, and energy consumption, respectively, and 5.3.4 summarizes these results and presents an analysis.

5.3.1 Execution Time

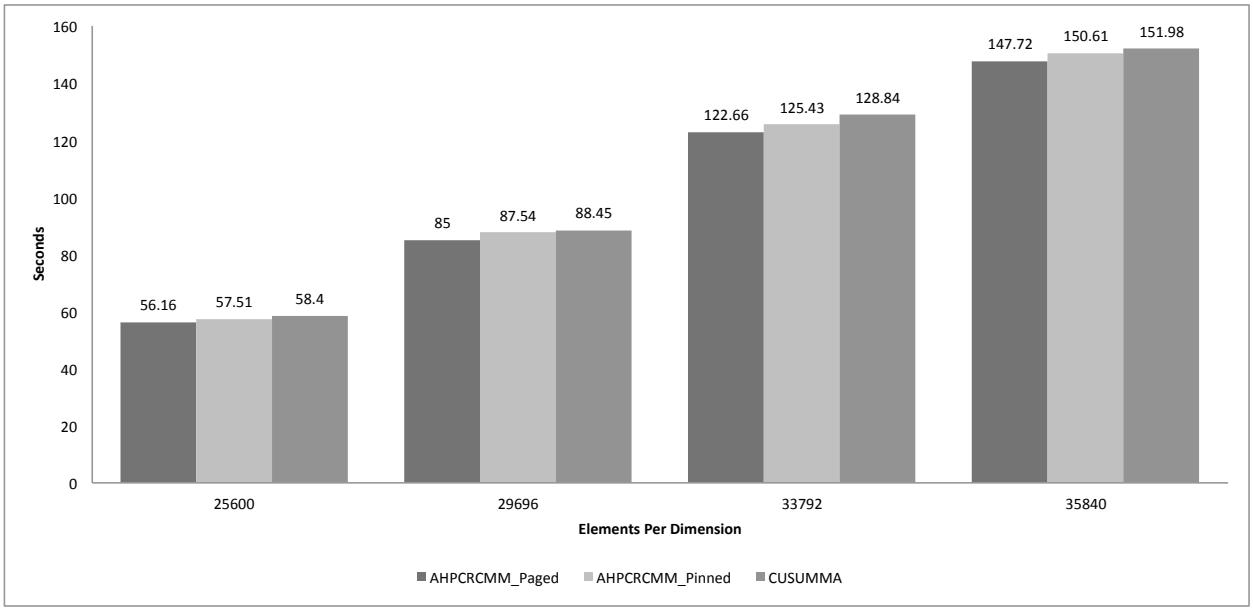

Figure 5.4.A – Class 2 (medium matrix) experiments: total execution time

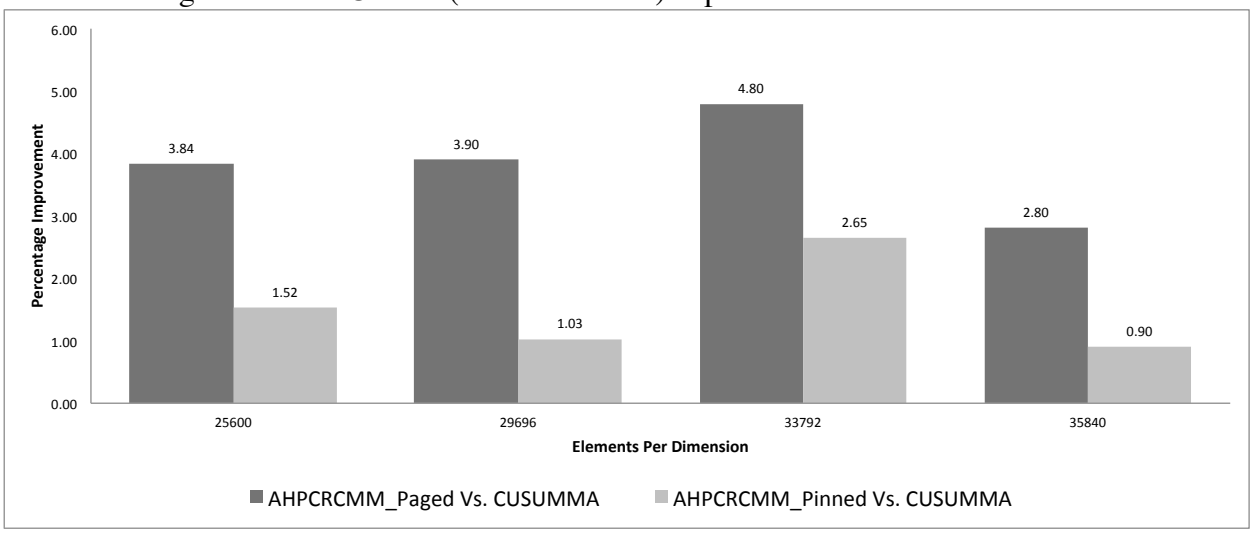

Figure 5.4.B – Class 2 (medium matrix) experiments: total execution-time improvement

Figure 5.4.1 presents the execution times of the Class 2 experiments. For the same set of experiments, Figure 5.4.2 presents the execution-time percentage improvement of AHPCRCMM routines (paged and pinned), as compared to that of CUSUMMA. Again, if the percentage improvement is negative (positive), it means the AHPCRCMM routine is slower (faster).

The differences in how CUSUMMA and AHPCRCMM, which are mentioned above, likely explain the total execution-time data presented in Figure 5.4.1. As shown in Figure 5.4.2, for Class 2 matrix sizes, AHPCRCMM always performs better than CUSUMMA, and AHPCRCMM_Paged performs better than AHPCRCMM_Pinned. The performance advantage attained by AHPCRCMM_Paged (AHPCRCMM_Pinned) over CUSUMMA ranges from 2.8% (.9%) to 4.8% (2.65%) across the matrix sizes used in the experiments.

The total execution- time performance improvement attained by AHPCRCMM_ paged over CUSUMMA is possibly due to three design decisions:

1. JIT data creation/deletion;
2. efficient host-to-device data transfer (explained in Chapter 3); and
3. avoidance of the creation and deletion of temporary block buffers for each host-to-device data transfer.

JIT data creation/deletion provides one explanation of the performance advantage of AHPCRCMM_Paged over CUSUMMA, as well as AHPCRCMM_Pinned. (Again, note that JIT data deletion is not possible when pinned memory in the host is employed (explained in Chapter 3). The other two design decisions likely provide performance advantages to both AHPCRCMM routines, allowing them to perform better than CUSUMMA. An additional reason for the better performance of AHPCRCMM_Pinned is the use of remote data memory transfer (RDMA), which enables data transfers (between host and device) to be performed concurrently with GPGPU kernel execution. This hides all data transfer latency between host and device, with the exception of the first host-to-device and last device-to-host data transfers.

5.3.2 Power Consumption

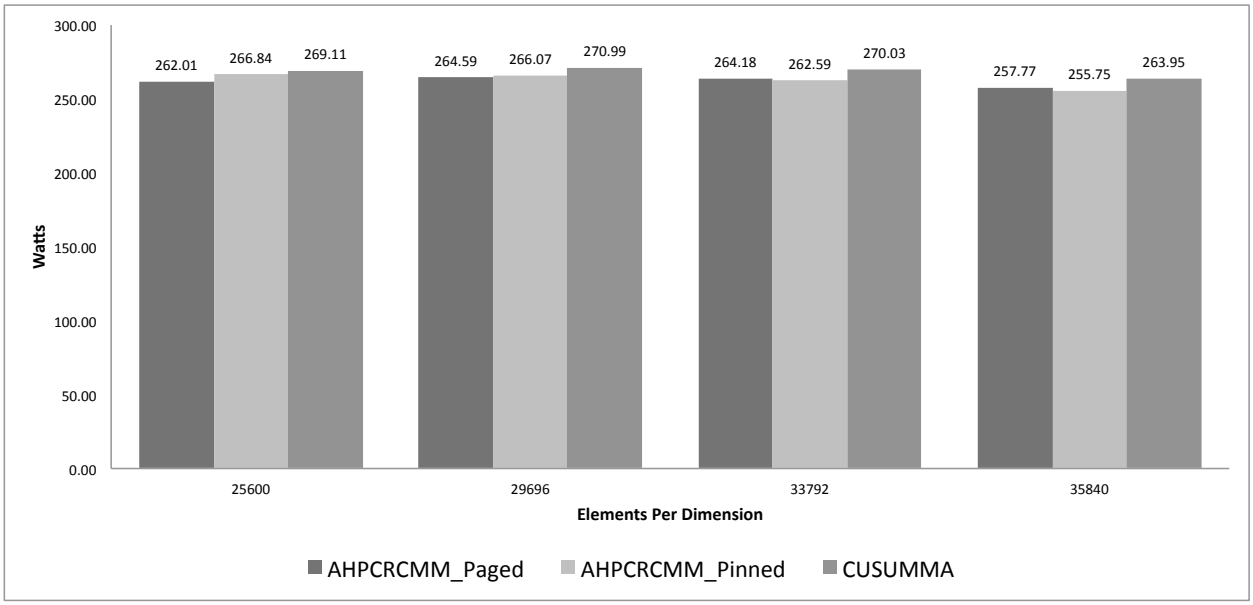

Figure 5.5.1 – Class 2 (medium matrix) experiments: total average power consumption

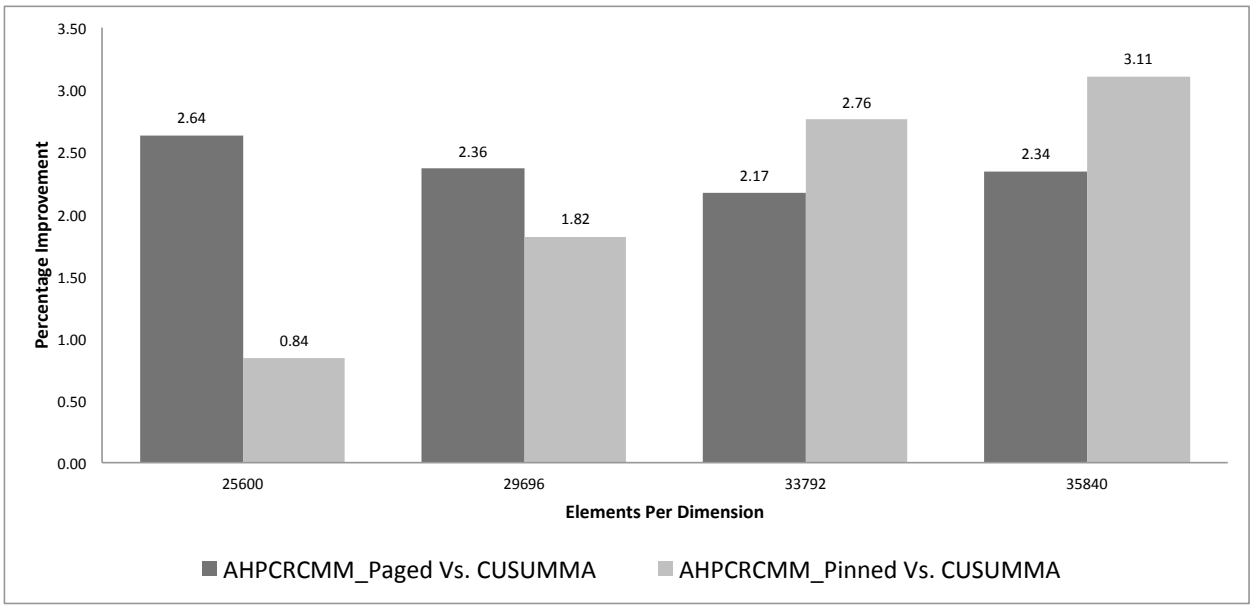

Figure 5.5.2 – Class 2 (medium matrix) experiments: total average power consumption improvement

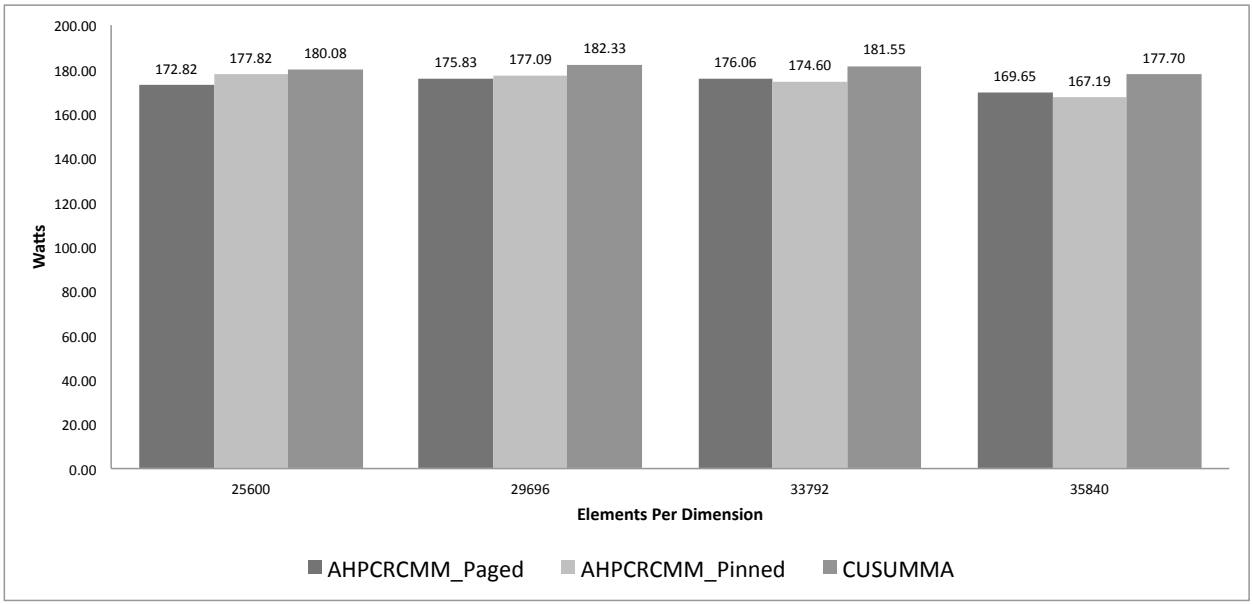

Figure 5.5.3 – Class 2 (medium matrix) experiments: GPGPU average power consumption

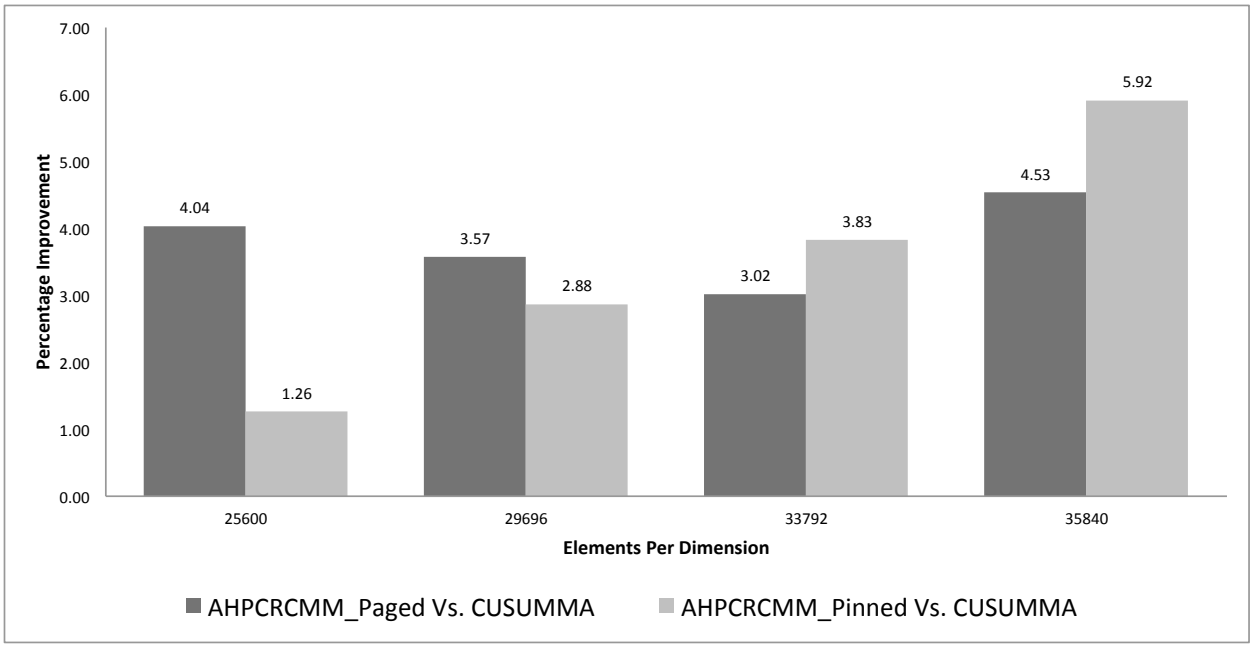

Figure 5.5.4 – Class 2 (medium matrix) experiments: GPGPU average power consumption improvement

For the Class 2 experiments, Figures 5.5.1 and 5.5.3 present the total average power consumption data and that attributable to only the GPGPU, respectively. For the same set of experiments, Figures 5.5.2 and 5.5.4 present the average power consumption improvement of the

AHPCRCMM routines (paged and pinned), as compared to that of CUSUMMA. Again, if the percentage improvement is negative (positive), it means the AHPCRCMM routine has higher/worse (lower/better) average power consumption.

The differences in steps 2, 3, and 4 of the AHPCRCMM (paged and pinned) routines as compared to the analogous steps of the CUSUMMA routine (described earlier in this section) are reflected in the total and GPGPU average power consumption data depicted in Figures 5.5.1 and 5.5.3, respectively. As can be seen, the AHPCRCMM routines consume less power than CUSUMMA. In terms of total average power consumption, the paged version consumes 2.17% to 2.64% less power and for the pinned version .84% to 3.11% less. Measuring the average power consumption for just the GPGPU, we see a 3.02% to 4.53% performance advantage for the paged version and a 1.25% to 5.92% for the pinned version. Also, it is important to note that on average over 66.5% of the total power consumption is attributable to the GPGPU.

We believe that the relatively small power performance advantages of the AHPCRCMM routines are due to two facts: (1) CUSUMMA transfers (in blocks) a total of three matrices from host to device, while AHPCRCMM routines only moves the two input matrices; and (2) CUSUMMA has two data movements from device to host memory, while the AHPCRCMM routines move this data once. CUSUMMA's additional data transfers between host and device translate to a greater GPGPU device memory footprint than that of the AHPCRCMM routines. The power performance advantage of the pinned version also is due to the fact that it transfers data blocks of matrices A and B during SGEMM kernel execution, thus, hiding the latency of the data movement from host to device.

An unexpected discovery is that for the largest matrix, i.e., 35,840 *x* 35,840, regardless of the routine driving the experiment, the average GPGPU power consumption decreases and, as is shown in Section 5.3, this continues to be the case as we experiment with even larger matrices. Currently, we do not know why this is the case – this is left for future work.

5.3.3 Energy Consumption

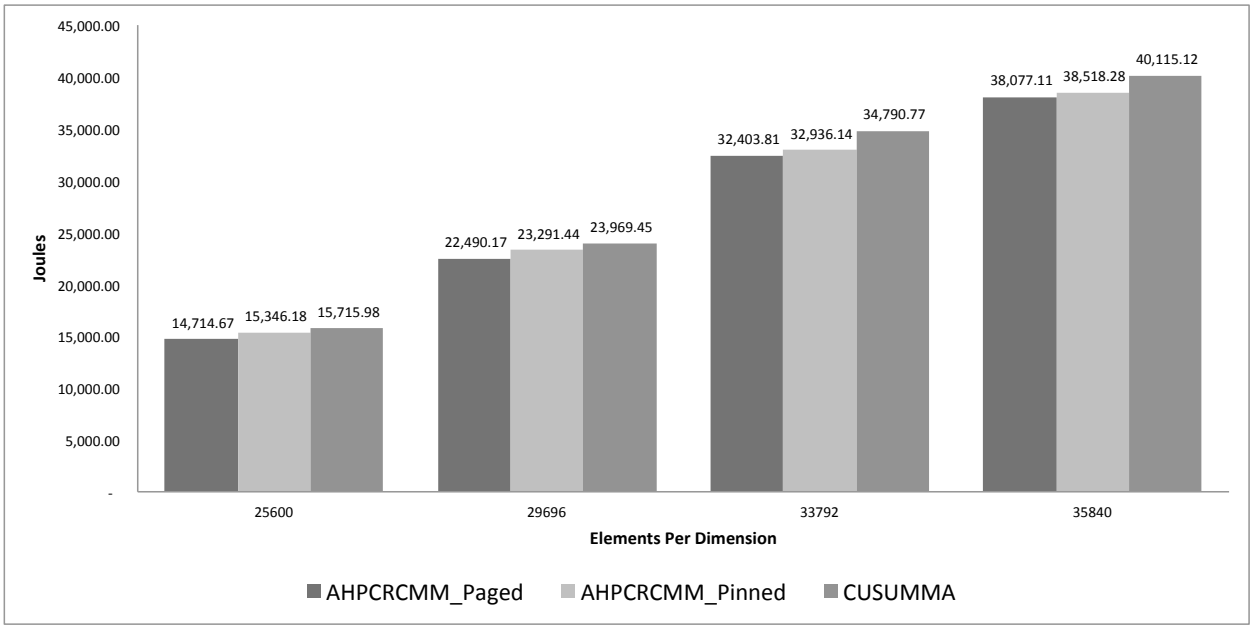

Figure 5.6.1 – Class 2 (medium matrix) experiments: total average energy consumption

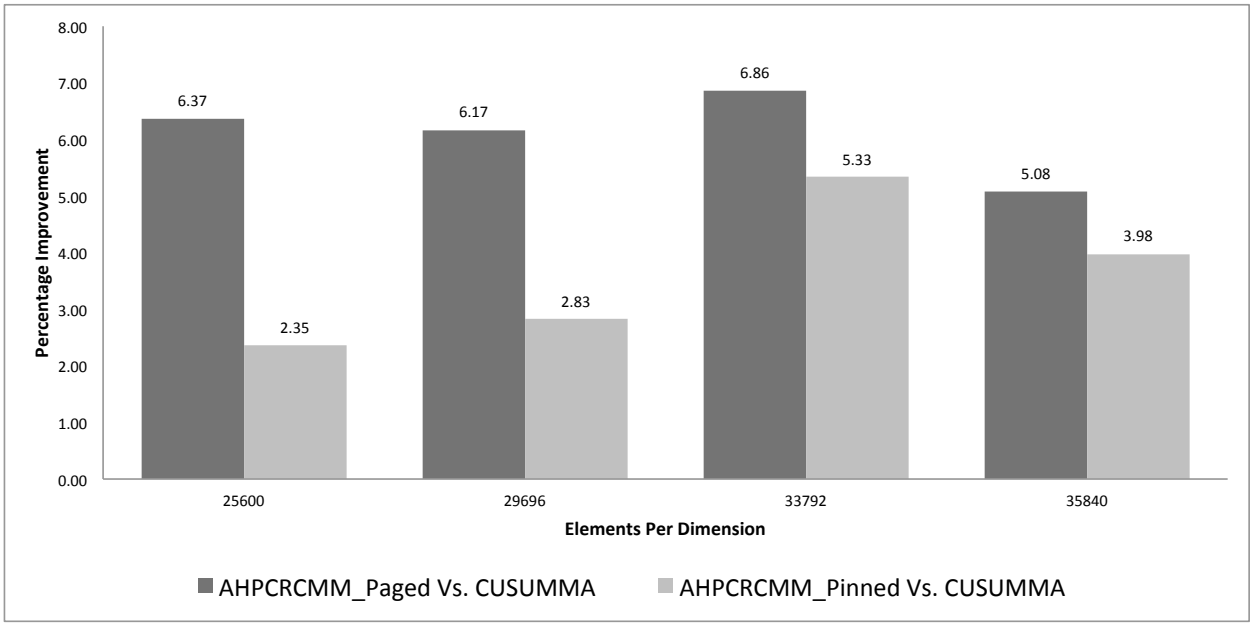

Figure 5.6.2 – Class 2 (medium matrix) experiments: total average energy improvement

Figure 5.6.1 presents the total average energy consumption results for Class 2 experiments, while Figure 5.6.2 presents the total average energy consumption percentage improvement of AHPCRCMM routines (paged and pinned), as compared with that of CUSUMMA. Again, if the

percentage improvement is negative (positive), it means the AHPCRCMM routine has higher/worse (lower/better) average energy consumption.

First, as expected for all experiments, we see that the energy consumption of CUSUMMA and AHPCRCMM (paged and pinned) increases with the matrix size. Also, we see that for all Class 2 experiments, the AHPCRCMM routines have better energy performance than does CUSUMMA, i.e., they consume from 2.35% to 6.86% less energy. AHPCRCMM_Paged (AHPCRCMM_ Pinned) exhibited a total average energy consumption improvement as compared to CUSUMMA between 5.08% and 6.86% (2.35% and 5.33%); this was largely due to an improvement in total execution-time performance between 2.8% and 4.8% (.9% and 2.65%) (across all Class 2 experiments) and between 2.1% and 2.64% (.8% and 3.1%) improvement in total average power consumption (across all Class 2 experiments).

To explore the effect of larger RAM capacity, we ran an additional experiment with the largest matrix size used in the Class 1 experiments, i.e., 21,504 *x* 21,504, on Test-bed 2. For this experiment, the total average energy consumption improvement of AHPCRCMM_ Paged (AHPCRCMM_ Pinned), as compared to CUSUMMA, was 7.4% (5.0%), due to an improvement of 5.9% (1.3%) in execution-time performance and a total average power consumption improvement of 1.0% (3.5%), these results can be found in Appendix A.

5.3.4 Summary

In the Class 2 experiments, AHPCRCMM_Paged always performs better than AHPCRCMM_Pinned (second best) and CUSUMMA in terms of execution time and energy consumption. The execution-time (energy consumption) advantage attained AHPCRCMM_Paged over CUSUMMA ranges from 2.8% (5.08%) to 4.8% (6.86%), whereas for AHPCRCMM_Pinned it ranges from .9% (2.35%) to 2.65% (5.33%). In terms of power performance, the AHPCRCMM routines perform better than CUSUMMA, the paged (pinned) version performing best for only the first (last) two

experiments of Class 2 but only by as much as 3.11%. The improvement in total (GPGPU) average power consumption of the AHPCRCMM routines ranges from 3% (1.25%) to 4.5% (5.92%). Thus, the energy performance advantage for the AHPCRCMM routines is a product of the execution-time and power performance advantages.

The performance improvement attained by AHPCRCMM over CUSUMMA is possibly due to three design decisions:

1. JIT data creation/deletion (AHPCRCMM_Paged);
2. efficient host-to-device data transfer (AHPCRCMM_Paged and AHPCRCMM_Pinned); and
3. avoidance of the creation and deletion of temporary block buffers for each host-to-device data transfer.

An additional reason for AHPCRCMM_Pinned, more noticeable when larger matrices are multiplied, is the use of remote data memory transfer (RDMA), which enables data transfers (between host and device) to be performed concurrently with GPGPU kernel execution, which accounts for over 66.5% of the total power consumption.

An unexpected discovery is that for the largest matrix, i.e., 35,840 *x* 35,840, regardless of the routine driving the experiment, the average GPGPU power consumption decreases and this continues to be the case as we experiment with even larger matrices. Currently, we do not know why this is the case – this is left for future work.

5.4 Performance Results For Class 3 (Large Matrix) Experiments

The experiments belonging to Class 3 employ matrix sizes such that no single matrix fits in GPGPU memory; in the case of our experimental platforms this is 5.2GB. Thus, only CUSUMMA and AHPCRCMM drive experiments in this class. For the three matrix sizes employed in Class 3 experiments: 37,888, 41,984, and 46,080 AHPCRCMM operates differently than CUSUMMA. The process employed by AHPCRCMM is described in Section 3.1.2, while that employed by CUSUMMA

is described in Section 2.2.2. To multiply matrices of the sizes used in Class 3 experiments, both CUSUMMA and AHPCRCMM follow a similar general five-step process, i.e.:

1. At the host, tile input matrices A and B, and output matrix C, into blocks.
2. Transfer a block from each input matrix from host memory to GPGPU device memory.
3. Execute the CUDA SGEMM kernel on the GPGPU device.
4. Return to host memory the partial result, i.e., the block of C.
5. Free the associated host and GPGPU device memory.

Repeat steps 2 through 4 until all blocks of matrices A and B are processed.

In Class 3 experiments, we study the performance of the multiplication 37,888 (size I), 41,984 (size J), and 46,080 (size K) single-precision floating-point matrices. Sections 5.4.1, 5.4.2, and 5.4.3 present data that is used to analyze performance in terms of execution time, power consumption, and energy consumption, respectively, and 5.4.4 summarizes these results and presents an analysis.

Note that 46,080 *x* 46,080 is the largest size of square matrices that AHPCRCMM Solver_2 can multiply on Test-bed 1.

5.4.1 Execution Time

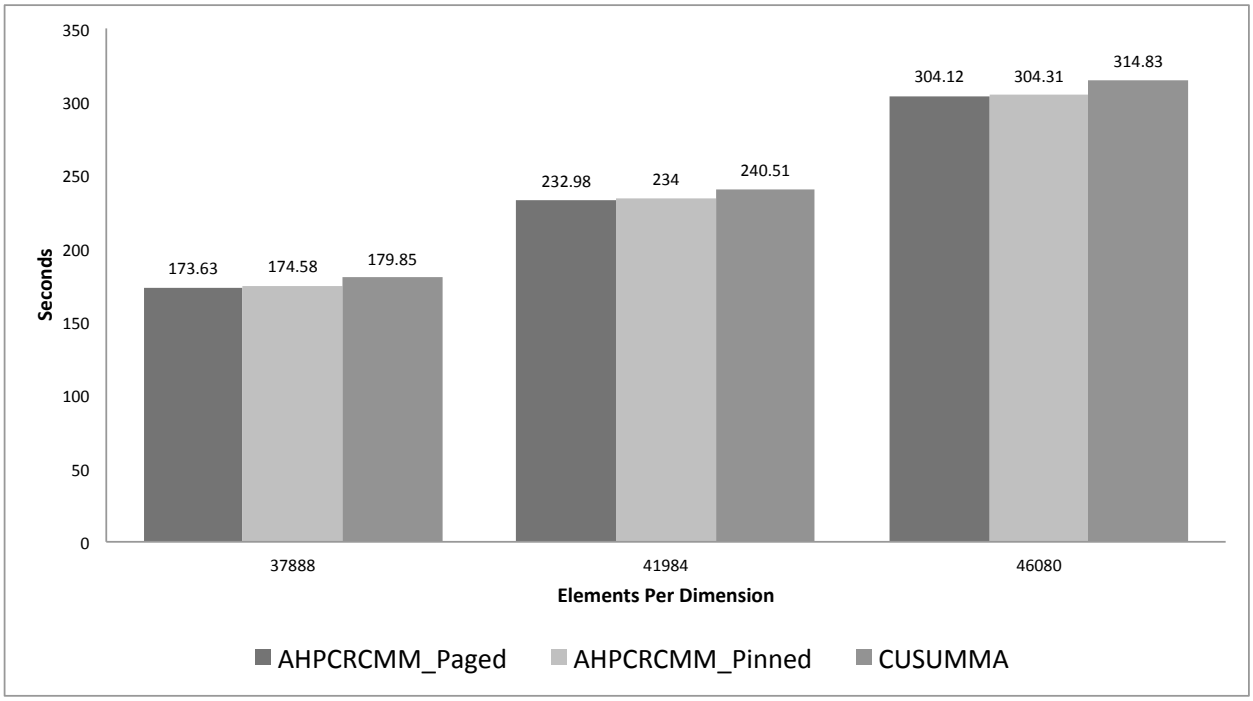

Figure 5.7.1 – Class 3 (large matrix) experiments: total execution time

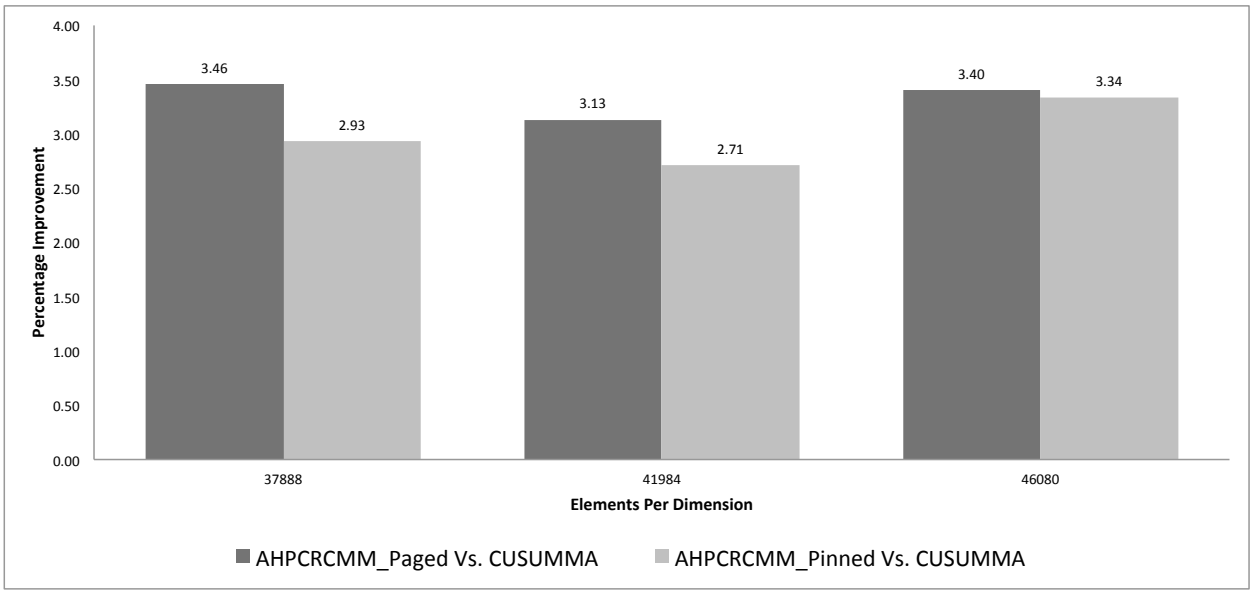

Figure 5.7.2 – Class 3 (large matrix) experiments: total execution-time improvement

Figure 5.7.1 presents the total execution time for AHPCRCMM_Paged, AHPCRCMM_Pinned, and CUSUMMA for experiments in Class 3. For the same set of experiments, Figure 5.7.2 presents the

execution-time percentage improvement of AHPCRCMM (paged and pinned), as compared to that of CUSUMMA. Again, if the percentage improvement is negative (positive), it means the AHPCRCMM routine is slower (faster).

The data depicted in Figure 5.7.1 shows that AHPCRCMM_Paged performs best in terms of execution time, followed by AHPCRCMM_Pinned, and then CUSUMMA. The total execution-time performance improvement of AHPCRCMM_Paged (AHPCRCMM_Pinned) in comparison to CUSUMMA is between 3.13% and 3.46% (2.71% and 3.34%) across all Class 3 experiments. This relatively small performance advantage is due to: (1) efficient data reuse of matrices A and B (explained in Section 3.2.2 Chapter 3); (2) avoidance of creating and freeing temporal block buffer of matrix A, which CUSUMMA does on each host-to-device transfer; and (3) work-sharing between the host (CPU) and GPGPU (explained in Section 3.3.3). Additionally, AHPCRCMM_Pinned hides the data transfer latency by concurrently transferring data between the host and device while the kernel is executing on the device. In conclusion, for the Class 3 experiments, AHPCRCMM_Paged and AHPCRCMM_Pinned provide better execution-time performance than CUSUMMA – they provide a 3-4% and 6-7% total execution-time improvement, respectively.

5.4.2 Power Consumption

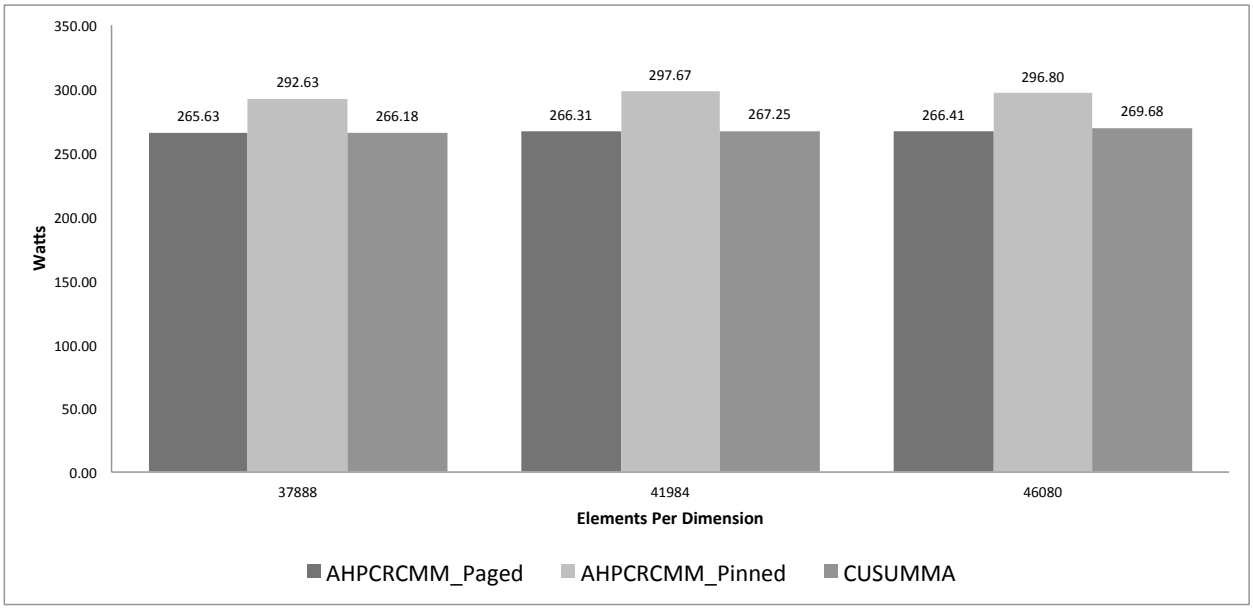

Figure 5.8.1 – Class 3 (large matrix) experiments: total average power consumption

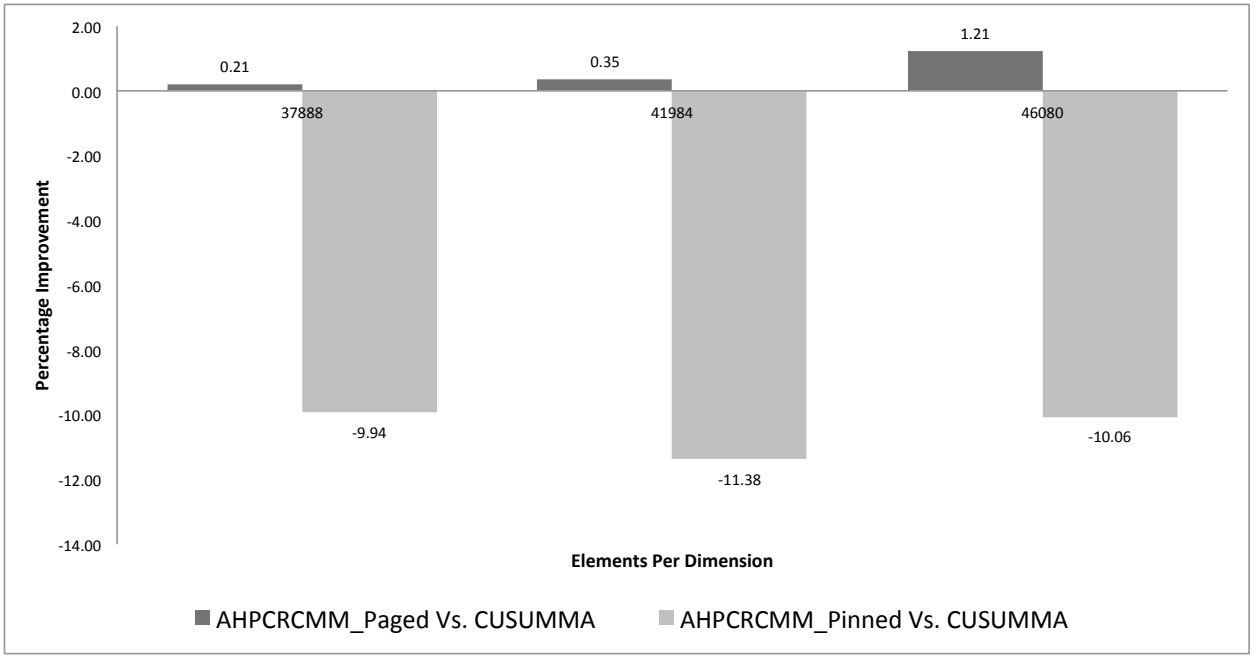

Figure 5.8.2 – Class 3 (large matrix) experiments: total average power improvement

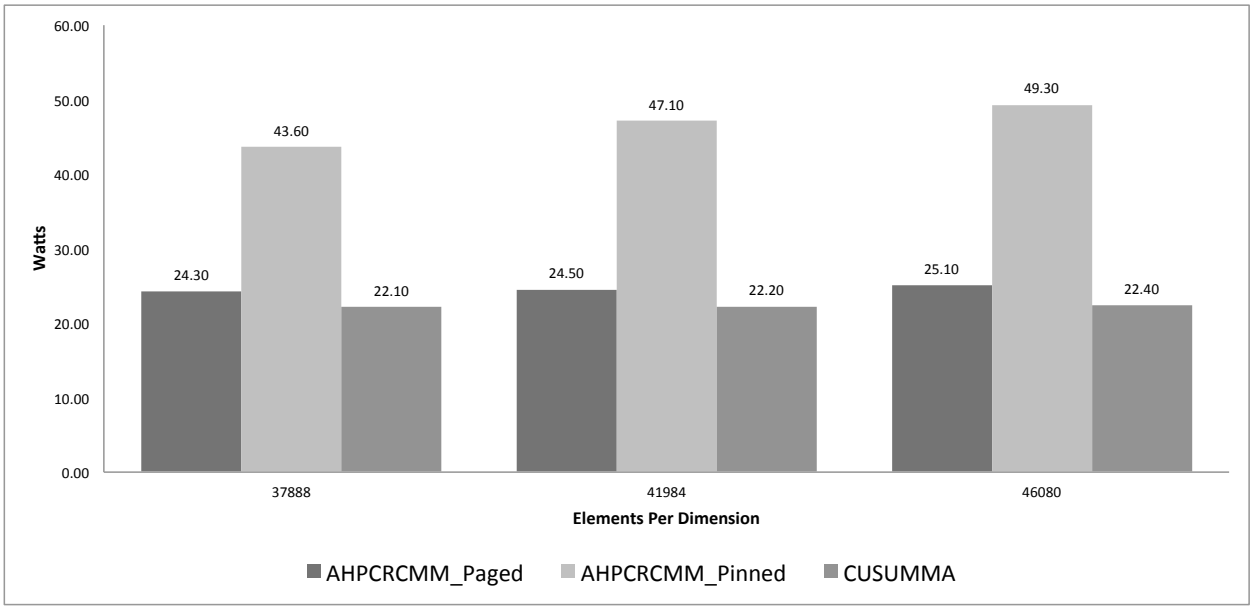

Figure 5.8.3 – Class 3 (large matrix) experiments: CPU average power consumption

Figures 5.8.1 and 5.8.3 present the total and CPU average power consumption for the Class 3 experiments, respectively. For the same set of experiments, Figure 5.8.2 presents the total average power consumption improvement of the AHPCRCMM routines (paged and pinned) as compared to CUSUMMA. Again, if the percentage improvement is negative (positive) it means that the AHPCRCMM routine has higher/worse (lower/better) average power consumption.

Figures 5.8.1 and 5.8.2 indicate that AHPCRCMM_Paged performs slightly better than CUSUMMA, and both perform much better than AHPCRCMM_Pinned, which for all matrix sizes consumes approximately 30 Watts more power on average, a 9.94% to 11.38% increase in total power consumption.

Because the AHPCRCMM routines, when working on the matrix sizes associated with Class 3 experiments, share work between the CPU and GPGPU (explained in Section 3.3.3), as compared to CUSUMMA, AHPCRCMM_Paged realizes a 9-12% higher average CPU power consumption but a 1-3% (data from Appendix C.2) lower average GPGPU power consumption. This is due to the fact that AHPCRCMM_Paged has efficient data reuse of matrices A and B (explained in Chapter 3).

In contrast, as shown in Figure 5.8.3, AHPCRCMM_Pinned consumes almost double the CPU power than do both AHPCRCMM_Paged and CUSUMMA. The work-sharing technique used by the AHPCRCMM routines iteratively adds the partial result, which is transmitted from the GPGPU device to the host, to the result container matrix C on the host. Although this methodology is used by both AHPCRCMM routines and it is used by them when multiplying the matrices of the Class 3 experiments, it is only in the Class 3 experiments that we observe this relatively large increase in CPU average power consumption and, thus, total power consumption. It appears that power consumption increases as the number of operations in the pinned host memory increases – staying rather constant across the three Class 3 experiments. Further investigation of this phenomenon is warranted, but it is left for future work.

In summary, for the matrix sizes used in the Class 3 experiments, AHPCRCMM_Paged performs slightly better than CUSUMMA in terms of power performance, and both perform much better than AHPCRCMM_Pinned, which for all matrix sizes consumes approximately 30 Watts more power on average – a 9.94% - 11.38% increase in power consumption – likely due to the high interaction between the CPU and pinned host memory, which is the case for larger matrices.

5.4.3 Energy Consumption

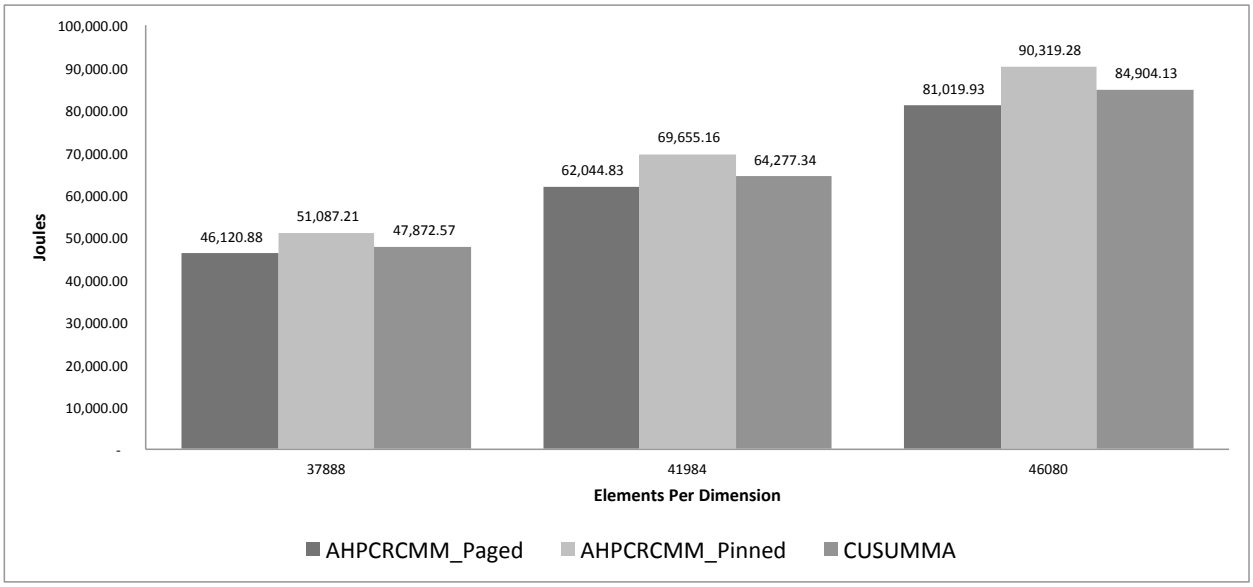

Figure 5.9.1 – Class 3 (large matrix) experiments: total average energy consumption

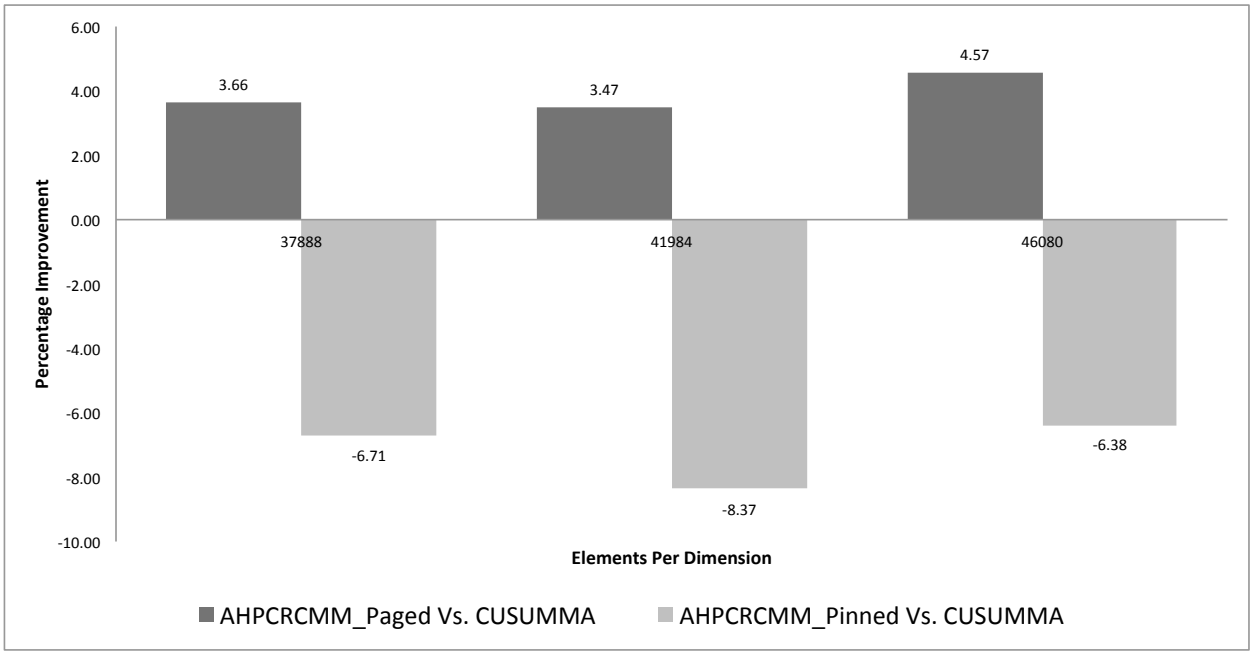

Figure 5.9.2 – Class 3 (large matrix) experiments: total average energy consumption improvement

Figure 5.9.1 presents the total average energy consumption of the Class 3 experiments, while Figure 5.9.2 presents the average energy consumption percentage improvement of the AHPCRCMM routines (paged and pinned), as compared to CUSUMMA. Again, if the percentage improvement is

negative (positive), it means the AHPCRCMM routine has higher/worse (lower/better) average energy consumption.

In comparison to CUSUMMA, AHPCRCMM_Paged consumes less energy – across all three experiments, the total average energy consumption improvement is between 3.66% and 4.57%. In contrast, AHPCRCMM_Pinned consumes more energy than CUSUMMA – across all three experiments, its total average energy consumption is greater by -6.38% to -8.37%. The performance of AHPCRCMM_Paged is not surprising since its execution times differ little from those of CUSUMMA, i.e., over all three experiments it exhibited an improvement in total execution-time performance between 3.13% and 3.46% (presented in Section 5.4.1) and had comparable power consumption, i.e., between .21% and 1.21% total average power consumption (presented in Section 5.4.2). The higher energy consumption of AHPCRCMM_Pinned is clearly due to its higher power consumption. AHPCRCMM_Pinned had execution times similar to those of CUSUMMA, i.e., it had an improvement in total execution time between 2.71% and 3.34% (presented in Section 5.2.3.1), but it consumed between 9.94% and 11.38% higher total average power (presented in Section 5.2.3.2). Furthermore, we noticed an improvement in total average energy consumption for the largest matrix sizes multiplied Test-bed 1 (46,080 elements per dimension) when multiplied on Test-bed 2 (see Appendix A.3). The improvement was 7.6%, which is due to an improvement of 9.1% in execution-time performance, regardless of the 2.2% higher total average power consumption. Using the largest matrix size CUSUMMA can solve for (13,282 elements per dimension), we noticed a total average energy improvement of 7.3%, which is due to an improvement of 6.8% in execution-time performance, and comparable (less than 0.1% difference) total average power consumption, these results can be found in Appendix A.

In addition, we noticed comparable total average energy consumption for the largest Class 3 matrix (46080 elements per dimension) when multiplied on Test-bed 2 (see Appendix A.3). The

improvement of 1.2% was due to an improvement of 13.2% in execution-time performance, regardless of 13.7% higher total average power consumption. For the largest matrix size that CUSUMMA can solve for ((59,008 elements per dimension), we observed for AHPCRCMM_Pinned a total average energy decrease of 2.0%, due to an improvement of 12.7% in execution-time performance, regardless of a 17.0% higher total average power consumption. These results can be found in table Appendix A.1, Appendix A.2, and Appendix A.3.

5.4.3 Summary

Results of the Class 3 experiments show that AHPCRCMM_Paged performs best in terms of execution time, followed by AHPCRCMM_Pinned and then CUSUMMA. However, the difference in the execution times of the AHPCRCMM routines and those of CUSUMMA are small, between 2.71% and 3.46%. Given the comparable power consumption of AHPCRCMM_Paged and CUSUMMA, i.e., an average (across the three experiments) of 0.5% total average power consumption, it is not surprising that AHPCRCMM_Paged attained an average energy performance improvement (across all three experiments) of 3.8%.

Given that AHPCRCMM_Pinned for all Class 3 matrix sizes consumes approximately 30 Watts more power than both AHPCRCMM_Paged and CUSUMMA, i.e., a 9.94% to 11.38% increase in power consumption, it is not surprising that it has poorer energy performance than both, e.g., an average (across all three experiments) of 7.1% higher than CUSUMMA.

For Class 3 experiments AHPCRCMM outperforms CUSUMMA in terms of execution time due to the following reasons: (1) JIT data creation/deletion in the initialization phase; (2) efficient data reuse of matrices A and B; and (3) avoidance of creating and freeing temporal block buffer of matrix A (which CUSUMMA does on each host-to-device transfer provide lower computation phase execution times) and work-sharing between the host (CPU) and GPGPU. Additionally, AHPCRCMM_Pinned

hides the data transfer latency by concurrently transferring data between the host and device during kernel execution.

It is interesting to note that because work sharing is employed by both AHPCRCMM routines when working with the matrix sizes associated with Class 3 experiments, a relatively large increase in average CPU and total power consumption was observed only for the larger matrix sizes used in Class 3 experiments executed by the AHPCRCMM_Pinned routine. Average CPU power consumption was 9-12% higher for AHPCRCMM_Paged but almost double for AHPCRCMM_Pinned (as compared to both AHPCRCMM_Paged and CUSUMMA). For AHPCRCMM_Paged, this is tempered by the constant 1-3% improvement in average power consumption of the GPGPU device for all Class 3 experiments (see Appendix C.3), which is due to efficient data reuse of matrices A and B). For AHPCRCMM_Pinned, it appears that power consumption increases with the number of operations in pinned host memory – staying rather constant across the three Class 3 experiments – and the CPU performs extra work when there is high interaction with pinned host memory. Further investigation of this phenomenon is warranted, but it is left for future work.

5.5 Summary of Experimental Results

CUBLAS is studied only in the Class 1 experiments, where input matrix sizes such that matrices A, B, and C together fit in GPGPU memory. Since CUBLAS, CUSUMMA, and AHPCRCMM employ the CUDA SGEMM kernel for matrices of these sizes and these sizes are relatively small compared to the sizes of the matrices multiplied in Class 2 and Class 3 experiments, their execution-time, power, and energy performance are not significantly different for the Class 1 experiments. For all four Class 1 experiments, the AHPCRCMM routines execute faster than CUBLAS, the paged version performing slightly better than the pinned version. However, there is no clear winner in terms of power or energy performance.

For the three largest matrix sizes, the execution-time improvement is 1-5%. In contrast, an improvement of 13% is attained for the smallest matrix size, which is due to one significant difference in the design of the AHPCRCMM routines, i.e., JIT data creation in the initialization phase, the effect of which increases as the matrix size decreases. For this reason the AHPRCMM routines also perform better than CUSUMMA in the Class 1 experiments for all matrices except 17,408 *x* 17,408. The difference in performance ranges from -2.47% (for this matrix size) to 4.67%. Although the differences in power consumption (across all four routines) are less than 2.11%, energy performance differences, which range from -4.36% to 12.88% improvement, do not correlate with the execution-time differences except in the case of the smallest matrix size.

Class 2 and Class 3 experiments provide a comparison of the AHPCRCMM routines and CUSUMMA.

For all these experiments, AHPCRCMM_Paged always performs better than AHPCRCMM_Pinned (second best) and CUSUMMA in terms of execution-time performance. However, the performance advantage is less than 5%.

In the Class 2 experiments, AHPCRCMM_Paged always performs better than AHPCRCMM_Pinned (second best) and CUSUMMA in terms of both execution time and energy consumption. The energy consumption advantage attained AHPCRCMM_Paged over CUSUMMA ranges from 5.08% to 6.86%, whereas for AHPCRCMM_Pinned it ranges from 2.35% to 5.33%. In terms of power performance, the AHPCRCMM routines perform better than CUSUMMA, the paged (pinned) version performing best for only the first (last) two experiments of Class 2 but only by as much as 3.11%. The improvement in total (GPGPU) average power consumption of the AHPCRCMM routines ranges from 3% (1.25%) to 4.5% (5.92%). Thus, the energy performance advantage for the AHPCRCMM routines is a product of the execution-time and power performance advantages.

The performance improvement attained by AHPCRCMM over CUSUMMA is possibly due to three design decisions:

1. JIT data creation/deletion (AHPCRCMM_Paged);
2. efficient host-to-device data transfer (AHPCRCMM_Paged and AHPCRCMM_Pinned); and
3. avoidance of the creation and deletion of temporary block buffers for each host-to-device data transfer.

An additional reason for AHPCRCMM_Pinned, more noticeable when larger matrices are multiplied, is the use of remote data memory transfer (RDMA), which enables data transfers (between host and device) to be performed concurrently with GPGPU kernel execution, which accounts for over 66.5% of the total power consumption.

In terms of power performance, the AHPCRCMM routines perform better than CUSUMMA in the Class 2 experiments -- better in terms of total (GPU) average power consumption by .84% (1.25%) to 3.11% (5.92%). However in the Class 3 experiments, AHPCRCMM_Pinned, for all matrix sizes, consumes approximately 30 Watts more power than both AHPCRCMM_Paged and CUSUMMA (which have comparable power performance), i.e., a 9.94% to 11.38% increase in power consumption. Since, in the Class 2 experiments, the AHPCRCMM routines enjoy both execution-time and power consumption performance advantages over CUSUMMA, they also enjoy an energy performance advantage, which ranges from 2.35% to 6.86%. Similarly, in Class 3 experiments, AHPCRCMM_Paged, with power performance comparable to that of CUSUMMA, attained an average energy performance improvement over CUSUMMA (across all three experiments) of 3.8%. In contrast, however, given that AHPCRCMM_Pinned had the highest power consumption for all three experiments, it has the poorest energy performance, e.g., an average (across all three experiments) of 7.1% higher than CUSUMMA.

The techniques used by the AHPCRCMM routines that could possibly be the reason for decreased execution time, power consumption, and energy consumption are:

- JIT data creation/deletion in the initialization phase, which only has a significant execution-time advantage for the smallest matrix experimented with;
- fewer host-to-device data transfers than CUSUMMA without creation and deletion of temporary block buffers, which reduces execution time and we believe also reduces GPGPU average power consumption (50% of the total average power consumption) due to a smaller GPGPU memory footprint;
- the use of remote data memory transfer (RDMA) by AHPCRCMM_Pinned, which enables data transfers (between host and device) to be performed concurrently with GPGPU kernel execution, thus, decreasing execution time and more noticeable when larger matrices are multiplied;
- efficient data reuse of matrices A and B; and
- work-sharing between the host (CPU) and GPGPU. It is interesting to note that although work sharing is employed by both AHPCRCMM routines, when working with the matrix sizes associated with Class 2 and Class 3 experiments, the relatively large increase in average CPU and total power consumption – 9-12% higher average CPU power consumption for AHPCRCMM_Paged and almost double for AHPCRCMM_Pinned – was observed only for the larger matrix sizes used in Class 3 experiments

An unexpected discovery is that for the largest matrices, i.e., 35,840 *x* 35,840, regardless of the routine driving the experiment, the average GPGPU power consumption decreases and this continues to be the case as we experiment with even larger matrices. Currently, we do not know why this is the case – this is left for future work.

Chapter 6: Conclusions and Future Work

This thesis employed software techniques to design and develop a new single-precision matrix-matrix multiplication routine for CPU/GPGPU node execution. Although we expected our routine to execute significantly faster and/or consume significantly less power than CUBLAS or CUSUMMA, this was not the case for our experiments conducted on Test-bed 1. Nonetheless, for the experiments conducted, our AHPCRCMM routine did achieve better execution-time performance for all matrix sizes except for one case and that performance was nearly equal – and, energy performance was comparable in all cases. Note, however, that the improvement in terms of execution time in all but one case, i.e., the smallest matrix used in our experiments, was less than 5%. Regardless of this, our routine is scalable, efficient in terms of the amount of data transferred, and, most importantly, can multiply larger matrices than either CUBLAS or CUSUMMA. Finally, for matrices in Class 3 and larger matrices, AHPCRCMM has a host memory requirement, rather than the GPGPU memory requirement of CUBLAS and CUSUMMA. This latter point is important since host memory is usually much larger than and is cheaper per byte than GPGPU memory and, thus, on CUDA-capable GPGPUs AHPCRCMM has the potential to multiply matrices of sizes that cannot be accommodated by GPGPU memory and CUSUMMA at comparable performance and/or lower power consumption.

In developing this routine, we learned quite a bit and we summarize these "lessons learned" below.

Pinned Memory: Although we thought that the pinned memory implementation (AHPCRCMM_Pinned) would yield better performance than the paged implementation (AHPCRCMM_Paged), this was not true for total execution time and total power consumption. However, AHPCRCMM_Pinned did yield better execution-time performance during the computation phase, which consumes more of the total execution time as the matrix size increases. The execution-time cost during the allocation and de-allocation of pinned memory is high, resulting in total execution-time

performance that is comparable to that of AHPCRCMM_Paged. In addition, the power consumption of the two implementations is comparable when there is no interaction between the host CPU and the data related to result matrix C. Furthermore, when there is interaction between the host CPU and this data, i.e., when the host CPU is performing adding a partial result of matrix C (generated by and transferred from the GPGPU) and the matrix C stored in host memory, we notice a 2x increase in host CPU average power consumption for the AHPCRCMM_Pinned in contrast to the AHPCRCMM_Paged.

Concurrent Data Transfer with Kernel Execution: The latest generation of GPGPU technology now supports concurrent data transfer between host and device with kernel execution. In this research, this technology was utilized and studied with a focus on improving execution-time performance and power consumption. Our research demonstrates that this technique becomes meaningful in terms of execution-time performance as the problem size increases.

Furthermore, we discovered that using this technique in combination with CPU/GPGPU work-sharing, where the host CPU has considerable interaction with allocated pinned memory, there are negative consequences in terms of the power consumption performance of the host CPU. We found this anomaly thanks to the capability of our test-bed to provide separate power measurements for multiple components of the computer system, i.e., the CPU, RAM, and GPGPU, as well as total system power, and analysis of their behavior for the various experiments conducted.

JIT Data Creation: We also utilized the JIT data creation technique, which is fully used by the paged version of AHPCRCMM. This technique allocates or de-allocates memory associated with matrices A, B, and C in the host during co-processor kernel execution, thus, hiding the latency of these operations and resulting in lower execution-time performance.

Additionally, for the largest class of matrices (Class 3 matrices), AHPCRCMM makes use of work-sharing by the host CPU and GPGPU, which enables the use of a pure outer-product approach to

transfer blocks of input matrices A and B, and output partial results of matrix C, thus, decreasing the total amount of data transmitted between host and device.

Host Memory Constraint: As mentioned above, for matrices in Class 3 and larger matrices, AHPCRCMM has a host memory requirement, rather than the GPGPU memory requirement of CUBLAS and CUSUMMA. This latter point is important since host memory is usually much larger than and is cheaper per byte than GPGPU memory and, thus, on CUDA-capable GPGPUs AHPCRCMM has the potential to multiply matrices of sizes that cannot be accommodated by GPGPU memory and CUSUMMA at comparable performance and/or lower power consumption.

6.1 Improvements

For this thesis, AHPCRCMM_Pinned does not make use of the JIT data deletion technique, which is used by the paged version. Still, this technique could be implemented for the pinned version by adding intelligence to the work-sharing function. The work-sharing function is executed by a slave thread, which adds a partial result coming from the GPGPU device to the result matrix C stored in host memory. When the thread finishes adding the partial result to matrix C, it should be safe to free the host memory allocation related to the blocks of input matrices A and B that were used to create the last partial result.

6.2 Future Work

In this study, one of our main focuses was to minimize the amount of data transmitted between the host and GPGPU device, and we succeeded when compared to the competition. However, it would be worthwhile to conduct a detailed study of the relationship between energy consumption during data transfers, with different data package sizes while keeping data volume transmission latency in the same range. This would enable us to make better decisions regarding the data package size to transmit between the host and device, and possibly reduce even further the total energy consumption of the AHPCRCMM routine.

Another area of improvement relates to the memory allocation size at the GPGPU device. Currently, our AHPCRCMM Solver_2 calculates the maximum number of elements that fit in GPGPU memory and then partitions input matrices A and B, and output matrix C based on that number. This aims to use most of GPGPU memory per partial-result iteration. Still, it may be useful to use a different methodology, where instead of using most of GPGPU memory per partial-result generation, we could use the least amount of memory. This could be accomplished using the pinned version of Solver_2, which has the potential to concurrently transfer data between host and device with kernel execution. The trade-off of transferring more small data packages between the host and device, but keeping GPGPU memory allocation low, could have the potential of reducing overall GPGPU power consumption without affecting execution-time performance.

As mentioned in Chapter 5, the CPU power consumption doubled for our Solver_2 pinned version, in comparison with Solver_2 paged and CUSUMMA routines. This anomaly was found thanks to the detailed power measurements provided by the test-bed developed as part of this research. Furthermore, it would be of great interest to study this anomaly by itself, as the use of pinned memory for concurrent data transfer and kernel execution is a performance optimization technique that is becoming common. And this technique has the potential to reduce the power efficiency of a parallel application.

Finally, regarding AHPCRCMM's host memory constraint, we plan to extend this study to quantify the potential of AHPCRCMM to multiply matrices of sizes on CUDA-capable GPGPUs AHPCRCMM that cannot be accommodated by GPGPU memory and CUSUMMA at comparable performance and/or lower power consumption. This will include a scalability study that includes execution on the threads of multi-core CPUs as well.

References

[1] Asano, Shuichi, Tsutomu Maruyama, and Yoshiki Yamaguchi. "Performance comparison of FPGA, GPU and CPU in image processing." *International Conference on Field Programmable Logic and Applications. FPL 2009.* IEEE, 2009.

[2] Bennemann, Christoph, et al. "Teraflops for games and derivatives pricing." *Wilmott magazine*, 36: 50-54, 2008.

[3] Bracken, E., et al. "Power aware parallel 3-D finite element mesh refinement performance modeling and analysis with CUDA/MPI on GPU and multi-core architectures." *IEEE Transactions on Magnetics*, 48(2): 335-338, 2012.

[4] Choi, Jaeyoung, David W. Walker, and Jack J. Dongarra. "PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers." *Concurrency: Practice and Experience*, 6(7): 543-570, 1994.

[5] Clint Whaley, R., Antoine Petitet, and Jack J. Dongarra. "Automated empirical optimizations of software and the ATLAS project." *Parallel Computing*, 27(1): 3-35, 2001.

[6] David, Howard, et al. "RAPL: memory power estimation and capping." *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. IEEE, 2010.

[7] Enos, Jeremy, et al. "Quantifying the impact of GPUs on performance and energy efficiency in HPC clusters." *2010 International Green Computing Conference*. IEEE, 2010.

[8] Esmaeilzadeh, Hadi, et al. "Looking back on the language and hardware revolutions: measured power, performance, and scaling." *ACM SIGARCH Computer Architecture News*, Vol. 39. No. 1. ACM, 2011.

[9] Galbraith, Byron, “Computational Modeling of Biological Neural Networks on GPUs: Strategies and Performance”. Master’s Theses (2009 -), Paper 61 (2010).

[10] Ge, Rong, et al. "Powerpack: Energy profiling and analysis of high-performance systems and applications." *IEEE Transactions on Parallel and Distributed Systems*, 21(5): 658-671, 2010.

[11] Govindaraju, Naga K., et al. "High performance discrete Fourier transforms on graphics processors." *Proceedings of the 2008 ACM/IEEE conference on supercomputing*. IEEE Press, 2008.

[12] Jiao, Y., et al. "Power and performance characterization of computational kernels on the gpu." *2010 IEEE/ACM International Conference on Green Computing and Communications (GreenCom)* and *International Conference on Cyber, Physical and Social Computing (CPSCom)*. IEEE, 2010.

[13] Lam, Monica D., Edward E. Rothberg, and Michael E. Wolf. "The cache performance and optimizations of blocked algorithms." *ACM SIGARCH Computer Architecture News*, Vol. 19, No. 2. ACM, 1991.

[14] Meyer, Carl. *Matrix analysis and applied linear algebra book and solutions manual*, Vol. 2. Siam, 2000.

[15] Nvidia, C. U. D. A. "Cublas library." *NVIDIA Corporation, Santa Clara, California*, (2013).

[16] Portillo, Ricardo, et al. "Power versus performance tradeoffs of GPU-accelerated backprojection-based synthetic aperture radar image formation." *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2011.

[17] Sahin, Cagri, et al. "Initial explorations on design pattern energy usage." *2012 First International Workshop on Green and Sustainable Software (GREENS)*. IEEE, 2012.

[18] Silberstein, Mark, et al. "Efficient computation of sum-products on GPUs through software-managed cache." *Proceedings of the 22nd annual international conference on supercomputing*. ACM, 2008.

[19] Steigerwald, B., and A. B. H. I. S. H. E. K. Agrawal. "Developing green software." *Intel White Paper* (2011).

[20] Suda, Reiji. "Investigation on the power efficiency of multi-core and GPU Processing Element in large scale SIMD computation with CUDA." *Green Computing Conference, 2010 International*. IEEE, 2010.

[21] Tölke, Jonas, and Manfred Krafczyk. "TeraFLOP computing on a desktop PC with GPUs for 3D CFD." *International Journal of Computational Fluid Dynamics*, 22(7):443-456, 2008.

[22] Van De Geijn, Robert A., and Jerrell Watts. "SUMMA: Scalable universal matrix multiplication algorithm." *Concurrency-Practice and Experience*, 9(4):255-274, 1997.

[23] Vázquez, Francisco, et al. "Improving the performance of the sparse matrix vector product with GPUs." *2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*. IEEE, 2010.

[24] Weaver, Vincent M., et al. "Measuring energy and power with PAPI." *41st International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, 2012.

[25] Yang, Zhiyi, Yating Zhu, and Yong Pu. "Parallel image processing based on CUDA." *2008 International Conference on Computer Science and Software Engineering,* Vol. 3. IEEE, 2008.

Appendix A

Test-bed 2: Power and Performance Measurements Results

Problem Class	Class 1				Class 2			
Matrix Size (MB)	1,764				4,900			
Elements Per Row/Column	21,504				35,840			
Execution Phase	Init	Compute	Clean up	Total	Init	Compute	Clean up	Total
AHPCRC_MM_Paged	7.63	36.48	0.27	44.38	13.1	157.61	0.71	171.44
AHPCRC_MM_Pinned	8.65	35.53	1.83	46.01	14.4	160.31	5.21	179.87
CUSUMMA	3.11	44.91	0.66	48.68	8.47	171.93	1.85	182.25
NVIDIA CUBLAS	12.6	34.86	0.95	48.42				

Table A.1: Execution-Time (seconds) performance for Class 1 and Class 2 experiments

Problem Class	Class 3							
Matrix Size (MB)	8,100				13,282			
Elements Per Row/Column	46,080				59,008			
Execution Phase	Init	Compute	Clean up	Total	Init	Compute	Clean up	Total
AHPCRC_MM_Paged	17.98	322.47	2.11	342.56	29.31	683.18	3.43	715.92
AHPCRC_MM_Pinned	25.18	293.67	8.25	327.10	43.15	613.23	13.66	670.56
CUSUMMA	14.71	359.11	3.11	376.93	25.51	738.14	5.11	768.76

Table A.2: Execution-time (seconds) performance for Class 3 experiments

Problem Class	Class 1	Class 2	Class 3	
Elements Per Row/Column	21,504	35,840	46,080	59,008
Matrix Size (MB)	1,764	4,900	8,100	13,282
AHPCRC_MM_Paged	321.67	325.43	335.53	330.39
AHPCRC_MM_Pinned	313.96	317.3	373.34	386.46
CUSUMMA	314.42	328.81	328.17	330.13
NVIDIA CUBLAS	308.19			

Table A.3: Total average power consumption (Watts)

Problem Class	Class 1	Class 2	Class 3	
Elements Per Row/Column	21,504	35,840	46,080	59,008
Matrix Size (MB)	1,764	4,900	8,100	13,282
AHPCRC_MM_Paged	14,275.71	55,791.72	114,939.16	236,532.81
AHPCRC_MM_Pinned	14,445.30	57,072.75	122,119.51	259,144.62
CUSUMMA	15,305.97	59,925.62	123,697.12	253,790.74
NVIDIA CUBLAS	14,922.56			

Table A.4: Total average energy consumption (joules)

Appendix B

Test-bed 1: Execution-time Performance per Phase of Execution

Problem Class	Class 1 (1)							
Matrix Size (MB)	324				676			
Elements Per Row/Column	9,216				13,312			
Execution Phase	Init	Compute	Clean up	Total	Init	Compute	Clean up	Total
AHPCRC_MM_Paged	2.75	2.61	0.25	5.61	2.92	7.61	0.52	11.05
AHPCRC_MM_Pinned	2.84	2.57	0.21	5.62	3.11	7.54	0.42	11.07
CUSUMMA	0.15	5.24	0.41	5.80	0.31	10.34	0.93	11.58
NVIDIA CUBLAS	3.09	2.51	0.85	6.45	3.51	7.45	0.14	11.10

Table B.1.1: Execution-time (seconds) performance for execution phases of Class 1 problems

Problem Class	Class 1 (2)							
Matrix Size (MB)	1,156				1,764			
Elements Per Row/Column	17,408				21,504			
Execution Phase	Init	Compute	Clean up	Total	Init	Compute	Clean up	Total
AHPCRC_MM_Paged	3.13	16.69	0.91	20.73	3.39	31.17	0.11	34.67
AHPCRC_MM_Pinned	3.42	16.61	0.72	20.75	3.74	31.12	1.11	35.97
CUSUMMA	0.54	19.57	0.14	20.25	0.81	35.34	0.22	36.37
NVIDIA CUBLAS	4.19	16.52	0.21	20.92	5.08	30.98	0.31	36.37

Table B.1.2: Execution-time (seconds) performance for execution phases of Class 1 problems

Problem Class	Class 2 (1)							
Matrix Size (MB)	2,500				3,364			
Elements Per Row/Column	25,600				29,696			
Execution Phase	Init	Compute	Clean up	Total	Init	Compute	Clean up	Total
AHPCRC_MM_Paged	3.74	52.26	0.16	56.16	4.13	80.68	0.19	85.00
AHPCRC_MM_Pinned	4.18	51.79	1.54	57.51	4.55	80.23	2.76	87.54
CUSUMMA	1.16	56.93	0.31	58.40	1.55	86.49	0.41	88.45

Table B.2.1: Execution-time (seconds) performance for execution phases of Class 1 problems

Problem Class	Class 2 (2)							
Matrix Size (MB)	4,356				4,900			
Elements Per Row/Column	33,792				35,840			
Execution Phase	Init	Compute	Clean up	Total	Init	Compute	Clean up	Total
AHPCRC_MM_Paged	4.59	117.84	0.23	122.66	4.86	142.61	0.25	147.72
AHPCRC_MM_Pinned	5.29	117.46	2.68	125.43	5.28	142.19	3.14	150.61
CUSUMMA	2.12	126.21	0.51	128.84	2.25	149.15	0.58	151.98

Table B.2.2: Execution-time (seconds) performance for execution phases of Class 2 problems

Problem Class	Class 3 (1)							
Matrix Size (MB)	5,476				6,724			
Elements Per Row/Column	37,888				41,984			
Execution Phase	Init	Compute	Clean up	Total	Init	Compute	Clean up	Total
AHPCRC_MM_Paged	4.89	168.33	0.41	173.63	5.51	226.94	0.53	232.98
AHPCRC_MM_Pinned	6.45	164.85	3.28	174.58	7.41	222.18	4.41	234.00
CUSUMMA	2.53	176.68	0.64	179.85	3.11	236.58	0.82	240.51

Table B.3.1: Execution-time (seconds) performance for execution phases of Class 3 problems

Problem Class	Class 3 (2)			
Matrix Size (MB)	8,100			
Elements Per Row/Column	46,080			
Execution Phase	Init	Compute	Clean up	Total
AHPCRC_MM_Paged	6.15	297.31	0.66	304.12
AHPCRC_MM_Pinned	8.42	291.16	4.73	304.31
CUSUMMA	3.75	310.14	0.94	314.83

Table B.3.2: Execution-time (seconds) performance for execution phases of Class 3 problems

Appendix C

Test-bed 1: Total Average Power Consumption per System Component

Problem Class	Class 1			
Matrix Size (MB)	324	676	1,156	1,764
Elements Per Dimension (N)	9,216	13,312	17,408	21,504
AHPCRCMM_Paged	12.20	16.00	18.50	20.10
AHPCRCMM_Pinned	12.40	16.00	18.10	19.90
CUSUMMA	12.40	16.10	18.80	20.20
NVIDIA CUBLAS	12.20	16.10	18.70	20.30

Table C.1.1: Average power (Watts) of CPU for Class 1 problems

Problem Class	Class 2			
Matrix Size (MB)	2,500	3,364	4,356	4,900
Elements Per Dimension (N)	25,600	29,696	33,792	35,840
AHPCRCMM_Paged	21.10	21.50	21.90	22.00
AHPCRCMM_Pinned	21.60	22.30	22.60	23.10
CUSUMMA	20.90	21.60	21.80	21.80

Table C.1.2: Average power (Watts) of CPU for Class 2 problems

Problem Size	Class 3		
Matrix Size (MB)	5,476	6,724	8,100
Elements Per Dimension (N)	37,888	41,984	46,080
AHPCRCMM_Paged	24.30	24.50	25.10
AHPCRCMM_Pinned	43.60	47.10	49.30
CUSUMMA	22.10	22.20	22.40

Table C.1.3: Average power (Watts) of CPU for Class 3 problems

Problem Class	Class 1			
Matrix Size (MB)	324	676	1,156	1,764
Elements Per Dimension (N)	9,216	13,312	17,408	21,504
AHPCRCMM_Paged	171.24	171.98	170.67	170.22
AHPCRCMM_Pinned	169.40	169.90	170.37	174.44
CUSUMMA	171.75	170.58	168.53	169.42
NVIDIA CUBLAS	169.70	170.22	172.28	169.49

Table C.2.1: Average power (Watts) of GPGPU for Class 1 problems

Problem Class	Class 2			
Matrix Size (MB)	2,500	3,364	4,356	4,900
Elements Per Dimension (N)	25,600	29,696	33,792	35,840
AHPCRCMM_Paged	172.82	175.83	176.06	169.65
AHPCRCMM_Pinned	177.82	177.09	174.60	167.19
CUSUMMA	180.08	182.33	181.55	177.70

Table C.2.2: Average power (Watts) of GPGPU for Class 2 problems

Problem Size	Class 3		
Matrix Size (MB)	5,476	6,724	8,100
Elements Per Dimension (N)	37,888	41,984	46,080
AHPCRCMM_Paged	174.21	175.68	175.77
AHPCRCMM_Pinned	176.28	177.03	174.32
CUSUMMA	177.33	179.61	178.77

Table C.2.3: Average power (Watts) of GPGPU for Class 3 problems

Problem Class	Class 1			
Matrix Size (MB)	324	676	1,156	1,764
Elements Per Dimension (N)	9,216	13,312	17,408	21,504
AHPCRCMM_Paged	0.53	0.51	0.51	0.49
AHPCRCMM_Pinned	0.55	0.51	0.52	0.50
CUSUMMA	0.61	0.50	0.49	0.50
NVIDIA CUBLAS	0.52	0.53	0.50	0.49

Table C.3.1: Average energy (joules) of GPGPU for Class 1 problems

Problem Class	Class 2			
Matrix Size (MB)	2,500	3,364	4,356	4,900
Elements Per Dimension (N)	25,600	29,696	33,792	35,840
AHPCRCMM_Paged	0.47	0.47	0.27	0.48
AHPCRCMM_Pinned	0.48	0.48	0.47	0.49
CUSUMMA	0.49	0.47	0.48	0.48

Table C.3.2: Average energy (joules) of GPGPU for Class 2 problems

Problem Size	Class 3		
Matrix Size (MB)	5,476	6,724	8,100
Elements Per Dimension (N)	37,888	41,984	46,080
AHPCRCMM_Paged	0.48	0.47	0.47
AHPCRCMM_Pinned	0.49	0.49	0.49
CUSUMMA	0.47	0.48	0.46

Table C.3.3: Average energy (joules) of GPGPU for Class 3 problems

Appendix D

AHPCRCMM Code Listing

```
/*
**      AHPCRC Parallel Matrix-Matrix Multiply Routine Paged and Pinned versions.
**
**      This implementation work only with square matrices.
**
**      Enrique Portillo <eportillo2@miners.utep.edu>
**      Department of Computer Science
**      University of Texas at El Paso
**      12/16/2013
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <cuda.h>
#include <cublas.h>
#include <omp.h>
#include <memory>
#include <iostream>
#include <fstream>

#include <unistd.h>
#include <pthread.h>

using std::cout;
using std::endl;
using std::ifstream;

int SIZE ;
int BSIZE ;
//This code handle square matrices only.
int ADJUSTC;
int ADJUST;
int MP ;            //GPGPU multiprocessors
int GS ;  //Grid size of each dimension
int BC ;  //Device Block Size on the x direction (24 for this specific device C2075)
int WC ;//Width is the number of blocks per row which has to be W % 2 = 0
int B ;             //Device Block Size on the y direction (32 for this specific device C2075)
int W ;             //Width is the number of blocks per column which has to be W % 2 = 0
int  SN;  //Number of streams to use for the GPGPU execution
long int CB ;
int BLOCKSIZE;

int TWork;        //Tell the main thread other threads are still working
```

```
pthread_mutex_t MUTEX = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t PADD = PTHREAD_MUTEX_INITIALIZER;//Private Addition
//#define DEBUG 1
//#define PINNED //Uncomment this line for Pinned Version and re-compile.
struct Matrix
{
	float* block;
	Matrix()
	{
		#ifdef PINNED
			cudaError_t cError;
			cError = cudaHostAlloc( (void**) &block, BLOCKSIZE , cudaHostAllocDefault);
				if(cError!=cudaSuccess)
				{
					printf("Error Allocating Pinned Memory Memory at Matrix Struct\n");
				}
		#else
				block = (float*) malloc (BLOCKSIZE);
		#endif

	}
	Matrix(int a)
	{
		cudaMalloc((void**) &block,BLOCKSIZE);
	}

};
typedef struct {
   int    secs;
   int    usecs;
} TIME_DIFF;

struct TMat{
	float* M1;
	float* M2;
	float* M3;
	long int Q;
	cudaStream_t stream;
	cudaEvent_t event;
};

struct Dealloc{
	float* block;
	long int index;
	cudaEvent_t event;
};

void matrixMerge( Matrix** C, Matrix* TC, int row );//Merge Row to a Matrix
void matrixAdd(float* C, float* TC);// Add two matrices, store result in C
```

```
void matrixStrucInit(Matrix** M);// Initialize a Two dimensional set of Matrices (Each matrix is a block of elements)
void matrixStrucInit(Matrix** M,int value);// Initialize a Two dimensional set of Matrices (Each matrix is a block of elements)
void matrixInit(float* m);//Initializes a two dimensional Matrix with the values of counter
void matrixInit(float* m, int value);//Initializes a two dimensional Matrix with the values of counter
void copyRow(Matrix** Bb, Matrix* b, int irow,long int p);//Copy a Row of Blocks from a two dimensional set of blocks to a row of blocks
void copyBlock(Matrix** A ,Matrix* a,int irow, int icol,long int p,long int q);//Copy the contents of a Block from a Two dimensional Matrix to a block
void copyRow(Matrix** Bb, Matrix* b, int irow,long int p, cudaStream_t stream);//Copy a Row of Blocks from a two dimensional set of blocks to a row of blocks
void copyBlock(Matrix** A ,Matrix* a,int irow, int icol,long int p,long int q, cudaStream_t stream);//Copy the contents of a Block from a Two dimensional Matrix to a block
void printBlock(Matrix a);
void printBlock(float* a);
void echoD(const char* a);
void echo(const char* a);
TIME_DIFF * my_difftime (struct timeval *, struct timeval *);
int writeToFile(float* array, long int elemNum,char* name);
void randomInit(float *data, int size);
int readAndAllocate(float* container, int elemNum, char* name);


void matrixMerge( Matrix** C, Matrix* TC, int row )//Merge Row to a Matrix
{
	for(int i=0; i < WC ; i++)
	{
		matrixAdd( C[row][i].block, TC[i].block );
	}
}

void matrixAdd(float* C, float* TC)// Add two matrices, store result in C
{
#pragma omp parallel for schedule(dynamic)
	for(int i=0; i < B ; i++)		//Row
	{
		for(int k=0; k < BC ; k++)//Column
		{
			C[(i*BC)+k]+=TC[(i*BC)+k];
		}
	}
}
cudaStream_t Dstream;
void *matrixAddu(void* bundle)// Add two matrices, store result in C
{
	pthread_mutex_lock(&MUTEX);
	TWork++;
	pthread_mutex_unlock(&MUTEX);
```

```
	TMat* container=(TMat*)bundle;
	float* C=container->M1;
	float* TC=container->M3;
	long int q=container->Q;
#ifdef PINNED
	cudaEventSynchronize(container->event);
#endif
long int index=0;
	for(int i=0; i < q ; i++)
	{
		for(int k=0; k < CB ; k++)
		{
			index=(CB*i)+k;
			C[index]+=TC[index];
		}
	}
	pthread_mutex_lock(&MUTEX);
	TWork--;
	pthread_mutex_unlock(&MUTEX);
#ifdef PINNED
#endif
	pthread_exit(NULL);
}

void *DeallocAsync(void* bundle)// Add two matrices, store result in C
{
	Dealloc* container=(Dealloc*)bundle;
	cudaEventSynchronize(container->event);
	cudaFreeHost(container->block);
	pthread_exit(NULL);
}



void matrixStrucInit(Matrix** M)// Initialize a Two dimensional set of Matrices (Each matrix is a block of
elements)
{
#pragma omp parallel for schedule(dynamic)
	for(int i=0 ; i < W ; i++ )		//Move per Row
	{
		for(int k=0; k < WC ; k++)		//Move per Column
		{
			matrixInit( M[i][k].block );
		}
	}
}

void matrixStrucInit(Matrix** M,int value)// Initialize a Two dimensional set of Matrices (Each matrix is a block
of elements)
```

```
{
#pragma omp parallel for schedule(dynamic)
	for(int i=0 ; i < W ; i++ )
	{
		for(int k=0; k < WC ; k++)
		{
			matrixInit( M[i][k].block,value );
		}
	}
}

void matrixStrucInit(Matrix* M,int value)// Initialize a Two dimensional set of Matrices (Each matrix is a block of elements)
{
#pragma omp parallel for schedule(dynamic)
	for(int k=0; k < WC ; k++)
	{
		matrixInit( M[k].block,value );
	}
}

void matrixInit(float* m)//Initializes a two dimensional Matrix with the values of counter
{
	//float counter=0;
#pragma omp parallel for schedule(dynamic)
	for(int i=0; i < B ; i++)//Row
	{
		for(int k=0; k < BC ; k++)//Column
		{
			m[(i*BC)+k] = 1;//Initialize to 1
		}
	}
}

void matrixInit(float* m, int value)//Initializes a two dimensional Matrix with the values of counter
{
#pragma omp parallel for schedule(dynamic)
	for(int i=0; i < B ; i++)//Row
	{
		for(int k=0; k < BC ; k++)//Column
		{
			m[(i*BC)+k] = value;//Initialize to value
		}
	}
}

void copyRow(Matrix* Bb, Matrix* b, int irow, long int p)//Copy a Row of Blocks from a two dimensional set of blocks to a row of blocks
{
```

```
	cudaMemcpy(b[0].block, Bb[irow].block,sizeof(float)*p*CB, cudaMemcpyHostToDevice);
	free(Bb[irow].block);
}
cudaEvent_t Tevent;
void copyRow(Matrix* Bb, Matrix* b, int irow, long int p, cudaStream_t stream)//Copy a Row of Blocks from a
two dimensional set of blocks to a row of blocks
{
	cudaMemcpyAsync(b[0].block, Bb[irow].block,sizeof(float)*p*CB, cudaMemcpyHostToDevice, stream);
}

void copyBlock(Matrix** A ,Matrix* a,int irow, int icol, long int p, long int q)//Copy the contents of a Block from
a Two dimensional Matrix to a block
{
	cudaMemcpy(a[0].block, A[irow][icol].block,sizeof(float)*p*q, cudaMemcpyHostToDevice);
	free(A[irow][icol].block);
}


void copyBlock(Matrix** A ,Matrix* a,int irow, int icol, long int p, long int q, cudaStream_t stream)//Copy the
contents of a Block from a Two dimensional Matrix to a block
{
	cudaMemcpyAsync(a[0].block, A[irow][icol].block,sizeof(float)*p*q, cudaMemcpyHostToDevice,
stream);
}

void printBlock(Matrix a)
{
	for(int i=0; i < B ; i++)//row
	{
		for(int k=0; k < BC ; k++)//col
		{
			printf("\t%.2f\t|",a.block[(i*BC)+k]);
		}
		printf("\n");
	}
}

void printBlock(float* a)
{
	printf("\n-------------------------------------------------------------------------------\n");
	for(int i=0; i < B ; i++)//Row
	{
		for(int k=0; k < BC ; k++)//Column
		{
			printf("\t%.2f\t|",a[(i*BC)+k]);
		}
		printf("\n");
	}
	printf("-------------------------------------------------------------------------------\n");
```

```
}

void echo(const char* a)
{
	printf("\n%s\n",a);fflush(stdout);
}

void echoD(const char* a)
{
#ifdef DEBUG
	printf("\n%s\n",a);fflush(stdout);
#endif
}

TIME_DIFF * my_difftime (struct timeval * start, struct timeval * end)
{
   TIME_DIFF * diff = (TIME_DIFF *) malloc ( sizeof (TIME_DIFF) );

   if (start->tv_sec == end->tv_sec) {
      diff->secs = 0;
      diff->usecs = end->tv_usec - start->tv_usec;
   }
   else {
      diff->usecs = 1000000 - start->tv_usec;
      diff->secs = end->tv_sec - (start->tv_sec + 1);
      diff->usecs += end->tv_usec;
      if (diff->usecs >= 1000000) {
         diff->usecs -= 1000000;
         diff->secs += 1;
      }
   }

   return diff;
}

int MM() // Solver_2
{

#ifdef DEBUG

echo("Testing the Application");


#else


//Time variables
TIME_DIFF * difference;
TIME_DIFF * differenceCalc;
```

```
TIME_DIFF * differenceEnd;
struct timeval TVInitstart, TVInitend;
gettimeofday (&TVInitstart, NULL);

SN=2; //Number of streams to use for the GPGPU execution

long long int allocSize = (long long int)(BC*B)*WC*W*sizeof(float);
allocSize = allocSize / (1024*1024);

//Allocation Intelligence
int Maxw=1;

        printf("\n\n---No Matrix Fit in GPU Memory---\nOptimization Enabled - The algorithm will try to use
most of GPU Memory...\n");
long long int memSize=(long long int) 5 * 1024 * 1024 * 1024; // 5GB of memory for our Tesla C2075

long long int currentElements= (long long int)(BC*WC)*(B*Maxw) * (SN+1);        //Current Row Size in GPU
long long int TotalElements= (long long int)(BC*B)*WC*W ;                       //Total Matrix C Elements
long long int MaxElements= (long long int) (memSize/4);                         //Maximum ELements we can
have in GPU

printf("\nCurrentElements=\t%lld\nTotalElements=\t\t%lld\nMaxElements=\t\t%lld\n\n",currentElements,Total
Elements,MaxElements);

int T=1;
long int RA=WC*BC;
long int CA=W*B;
long int RB=W*B;
CB = WC*BC;                                     //Total elements in one dimension
long int EGMS =(long int)  MaxElements;
long int P = (EGMS - (SN*CB))/(1+CB);
if( P > ceil((1.0*CB)/T))
{
        P=ceil((1.0*CB)/T);
}

long int PCA = ceil((1.0*CA)/P);  //# Columns of Blocks of A - this could be interpreted as the number of columns
to solve
if(PCA>1)
{
        P=(1.0*CA/PCA);
        PCA = ceil((1.0*CA)/P);
}
long int X = P;
long int Q = (EGMS - X * CB)/(X + (SN*CB));
if( Q > RA)
{
        Q=RA;
}
```

```
long int PRA = ceil((1.0*RA)/Q);  //# Rows of Blocks of A
long int PRB = ceil((1.0*RB)/P);   //# Rows of Blocks of B

if(PRA>1)
{
        Q=(1.0*RA/PRA);
}

long int PRC = PRA;                          //# Rows of Blocks of C

long int AEGMS = (( Q*P + P * CB + SN*(Q*CB)) * sizeof(float)) /1024/1024; //Spaced used in GPU per
computation
printf("\nCA=\t\t%ld\n"
                "RA=\t\t%ld\n"
                "PCA=\t\t%ld\n"
                "PRA=\t\t%ld\n"
                "CB=\t\t%ld\n"
                "RB=\t\t%ld\n"
                "PRB=\t\t%ld\n"
                "EGMS=\t%ld MB\n"
                "P=\t\t%ld\n"
                "X=\t\t%ld\n"
                "Q=\t\t%ld\n"
                "AEGMS=\t%ld MB\n"
                "\n",CA,RA,PCA,PRA,CB,RB,PRB,(EGMS*sizeof(float))/1024/1024,P,X,Q,AEGMS);

BLOCKSIZE=(int)sizeof(float)*P*Q;
long int TP=P;
long int TQ=Q;

//Change global values if possible
//End allocation Intelligence

//return 0;

//cublasSgemm init Values -- A = m x k , B = k x n
char transa='n';
char transb='n';
int m=BC;
int n= BC * WC; //Size of B (in device) is a whole Row
int kk=B;
int lda=m;//m;          //Leading coordinate size for matrix A
int ldb=kk;//kk;        //Leading coordinate size for matrix B
int ldc=m;//m;          //Leading coordinate size for matrix C
//end cublasInitValues

//Initialize Matrices
Matrix** A;
```

```
Matrix* Bb;
Matrix* C;
Matrix* b;//Sub row of blocks of Bb

#ifdef PINNED
		cudaError_t cError;
		cError = cudaHostAlloc( (void**) &A,sizeof(Matrix*) * PRA , cudaHostAllocDefault);
			if(cError!=cudaSuccess)
			  {
				printf("Error Allocating Pinned Memory Memory at A\n");
				return 0;
			  }
		cError = cudaHostAlloc( (void**) &Bb,sizeof(Matrix) * PRB , cudaHostAllocDefault);
			if(cError!=cudaSuccess)
			  {
				printf("Error Allocating Pinned Memory Memory at Bb\n");
				return 0;
			  }
				C = (Matrix*) malloc( sizeof(Matrix) * PRC );
		cError = cudaHostAlloc( (void**) &b,sizeof(Matrix), cudaHostAllocDefault);
			if(cError!=cudaSuccess)
			  {
				printf("Error Allocating Pinned Memory Memory at b\n");
				return 0;
			  }

		echo("Allocating Host Memory - Pinned");
		printf("%lld MBs per Matrix (A,B,C)-> ",allocSize);

		TP=P;
		TQ=Q;
		BLOCKSIZE=sizeof(float)*TP*TQ;
		for(int i=0; i< PRA ; i++)
		{
			if(i+1==PRA && ((RA%Q)>0) && PRA>1 )
			{
				BLOCKSIZE=sizeof(float)*(P)*(RA%Q);
				TQ=(RA%Q);
			}
			A[i]=new Matrix[PCA];
			for(int k=0; k<PCA;k++)
			{
				std::uninitialized_fill_n(A[i][k].block, TP*TQ, 1);
			}
		}


		TP=P;
		for(int i=0; i< PRB ; i++)
```

```
		{
			if(i+1==PRB && (RB%P)>0 && PRB>1)
			{
				TP=(RB%P);
				printf("Allocating last small row of B\n");fflush(stdout);
			}
			Bb[i].block=(float*) malloc( sizeof(float)*CB*TP );
			std::uninitialized_fill_n(Bb[i].block, CB*TP, 1);

		}

			for(int i=0; i< PRC ; i++)
			{
				long int TQ=Q;
				if(i+1==PRC && (RA%Q)>0)
				{
					TQ=(RA%Q);
				}
				C[i].block=(float*) calloc ( CB*TQ ,sizeof(float));//Init of result matrix
			}

#else

		A = (Matrix**) malloc( sizeof(Matrix*) * PRA );
		Bb = (Matrix*) malloc( sizeof(Matrix) * PRB );
		C = (Matrix*) malloc( sizeof(Matrix) * PRC );
		echo("Allocating Host Memory - Pageable");
		printf("%lld MBs per Matrix (A,B,C)-> ",allocSize);fflush(stdout);

	srand(2006);

	// initialize host memory

		TP=P;
		TQ=Q;
		BLOCKSIZE=sizeof(float)*TP*TQ;
		for(int i=0; i< PRA ; i++)
		{
			if(i+1==PRA && ((RA%Q)>0) && PRA>1 )
			{
				BLOCKSIZE=sizeof(float)*(P)*(RA%Q);
				TQ=(RA%Q);
			}
			A[i]=new Matrix[PCA];
			for(int k=0; k<PCA;k++)
			{
				std::uninitialized_fill_n(A[i][k].block, TP*TQ, 1);
			}
		}
```

```
		TP=P;
		for(int i=0; i< PRB ; i++)
		{
			if(i+1==PRB && (RB%P)>0 && PRB>1)
			{
				TP=(RB%P);
				printf("Allocating last small row of B\n");fflush(stdout);
			}
			Bb[i].block=(float*) malloc( sizeof(float)*CB*TP );
			std::uninitialized_fill_n(Bb[i].block, CB*TP, 1);

		}

		BLOCKSIZE=CB*Q;
		for(int i=0; i< PRC ; i++)
		{
			if(i+1==PRC && (RA%Q)>0)
			{
				BLOCKSIZE=(int) CB*(RA%Q);
			}
			C[i].block=(float*) calloc ( BLOCKSIZE ,sizeof(float));//Init of result matrix
		}
		b = (Matrix*) malloc (sizeof(Matrix));

#endif

		//For Correctness purposes, please comment code below

		//readAndAllocate(A[0][0].block,(TP * TQ), "../A.gold" );
		//readAndAllocate(Bb[0].block,(CB * TP), "../B.gold" );

		//For Correctness purposes, please comment code above

printf("Done!\nInitialization of Matrices-> ");fflush(stdout);

//matrixStrucInit(A,1);   //Initiate all values of the internal Matrix to the second parameter

BLOCKSIZE=(int) sizeof(float)*P*Q;
printf("Here\n");fflush(stdout);
// Initiate device containers
float* temp;
#ifdef PINNED
Matrix *a= new Matrix[1];        //Sub block of A
cudaMalloc((void**) &temp, sizeof(float)*P*Q);
a[0].block = temp;
#else
Matrix *a= new Matrix[1];        //Sub block of A
```

```
cudaMalloc((void**) &temp, sizeof(float)*P*Q);
a[0].block = temp;
#endif

// Allocate and initialize an array of stream handles for Asynchronous instruction issue.
cudaStream_t *stream = (cudaStream_t *) malloc(SN * sizeof(cudaStream_t));    //Kernel executers
cudaStreamCreate(&Dstream);// DtoH global stream
for (int i = 0; i < SN; i++)
{
   cudaStreamCreate(&(stream[i]));
}

float** d_result=(float**)malloc(sizeof(float*) * SN);      //Result container at Device, note there is one result
container per Stream

BLOCKSIZE=CB*Q;
for (int i = 0; i < SN; i++)
{
        cudaMalloc((void**) &d_result[i],CB*Q * sizeof(float));//Because A=m*k B=k*n C=m*n && m=WC*BC
k=W*B n=WC*BC
        cudaMemset(d_result[i], 0, CB*Q * sizeof(float) );
}

float** result0 = (float**) malloc( sizeof(float*) * PRA );  // Result container in the Host, note there is one result
container per Stream
for(int i=0; i< PRA ; i++)
{

#ifdef PINNED
        cError = cudaHostAlloc( (void**) &result0[i],CB*Q * sizeof(float), cudaHostAllocDefault);//The result
holds a whole Row of blocks
        if(cError!=cudaSuccess)
                {
                        printf("Error Allocating Pinned Memory Memory at matrixM(Matrix,
Matrix)Error=%d\n",cError);
                        return 0;
                }
#else
        result0[i]=(float*) calloc ( CB*Q ,sizeof(float));
#endif
}

BLOCKSIZE=(int)CB*P*sizeof(float);
cudaMalloc((void**) &temp,CB*P*sizeof(float));
b[0].block = temp;


TMat* container=(TMat*) malloc( sizeof(TMat)* PRA);    //Package sent to a thread for host addition.
cudaEvent_t event[PRA];                                                    //Events to synchronize
```

```
kernel and DtoH memCopy
pthread_t threads[PRA];                                                    //Host threads to
perform addition step.
pthread_attr_t attr;                                                       //Pthread attributes
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
#pragma omp parallel for schedule(dynamic)
for(int e=0; e < PRA; e++)
{
        cudaEventCreate (&event[e]);
}

gettimeofday (&TVInitend, NULL);
difference = my_difftime (&TVInitstart, &TVInitend);

struct timeval TVCalcstart, TVCalcend;
gettimeofday (&TVCalcstart, NULL);

TP=P;
TQ=Q;
//int LastTQ=TQ;

printf("Start to solve the Matrix Multiply WC= %d W=%d BC=%d B=%d Matrix Size= %d x %d\n\nStart GPU
Calculation...\n",WC,W,BC,B, WC*BC , WC*BC);fflush(stdout);

int sc=0;
//int count=0;
#ifdef PINNED
copyBlock(A , a, 0, 0,TP,TQ, stream[0]);//Copy First block before we start
#else
copyBlock(A , a, 0, 0,TP,TQ);
#endif
cublasInit();

float totalSize=0;

for(int i=0; i < PRB ; i++)//Move per Row
{
        if(i+1==PRB && (RB%P)>0 && PRB>1)
        {
                printf("Changing values of CUBLASSGEMM...\n ");fflush(stdout);
                TP=(RB%P);
                //-- A = m x k , B = k x n
                m=TQ;//Reminder of RA
                n= CB;   //Size of B (in device) is a whole Row
                kk=TP;//Reminder of RB
                lda=kk;//m;             //Leading coordinate size for matrix A
                ldb=n;//kk;             //Leading coordinate size for matrix B
                ldc=n;//m;              //Leading coordinate size for matrix C
```

```
                //end cublasInitValues
        }else
        {
                printf("Hereee...\n ");fflush(stdout);
                TP=P;
                //-- A = m x k , B = k x n
                m=(TQ);//Reminder of RA
                n= CB;  //Size of B (in device) is a whole Row
                kk=(TP);//Reminder of RB
                lda=kk;//m;             //Leading coordinate size for matrix A
                ldb=n;//kk;             //Leading coordinate size for matrix B
                ldc=n;//m;              //Leading coordinate size for matrix C
                //end cublasInitValues
        }
        cudaDeviceSynchronize();
        printf("\n\tRow [%d] of %ld being computed...\n\n",i+1,PRB);fflush(stdout);
        #ifdef PINNED
                copyRow(Bb, b, i,TP, stream[0]);//Row copied by stream0
        #else
                copyRow(Bb, b, i,TP);
        #endif


        while(TWork>100)
        {
                printf("Too threat! %d\n",TWork);fflush(stdout);
                usleep(10000);
        }
        #pragma unroll
        for(int k=0; k < PRA ; k++)//Move per Column - Per A block in a row
        {
                sc=k%SN;//Decide which stream to use.
                printf("RowA[%d] of %ld \n ",k+1,PRA);fflush(stdout);
                long int TQ=Q;
                BLOCKSIZE= sizeof(float)*(CB)*(Q);
                if( k+1 == PRA && (RA%Q)>0)
                {
                        BLOCKSIZE=sizeof(float)*(CB)*(RA%Q);
                        TQ=(RA%Q);
                }
                printf("C - TP= %ld TQ= %ld \n ",TP,TQ);fflush(stdout);
#ifdef PINNED
                        cublasSetKernelStream(stream[sc]);              //Tell device which stream to use for
cublasSgemm()
                        cublasSgemm(transa, transb, n, m, kk, 1.0f, b[0].block, ldb, a[0].block, lda, 0.0f,
d_result[sc], ldc);//SGemm
                        cudaMemcpyAsync(result0[k], d_result[sc],sizeof(float) * CB * TQ,
cudaMemcpyDeviceToHost, stream[sc]);
                        cudaEventCreate (&event[k]);
```

```
                        cudaEventRecord (event[k], stream[sc]);//Keep track to know when to bring data back

                        container[k].M1 =C[k].block;
                        container[k].M3 =result0[k];
                        container[k].stream=stream[sc];
                        container[k].event=event[k];
                        container[k].Q=TQ;
                        pthread_create(&threads[k], &attr, matrixAddu,(void *) &container[k]);

#else
                        //cublasSetKernelStream(stream[sc]);            //Tell device which stream to use for
cublasSgemm()
                        cublasSgemm(transa, transb, n, m, kk, 1.0f, b[0].block, ldb, a[0].block, lda, 0.0f,
d_result[sc], ldc);//SGemm
                        float tempSize=0;
                        cudaMemcpy(result0[k], d_result[sc],sizeof(float) * CB * TQ ,
cudaMemcpyDeviceToHost);
                        tempSize=sizeof(float) * CB * TQ;
                        tempSize= tempSize/1024;//Size in KBs of data coming from GPGPU
                        tempSize= tempSize/1024;//Size in MBs of data coming from GPGPU
                        totalSize+=tempSize;

                        container[k].M1 =C[k].block;
                        container[k].M3 =result0[k];
                        container[k].stream=stream[sc];
                        container[k].Q=TQ;
                        pthread_create(&threads[k], &attr, matrixAddu,(void *) &container[k]);

#endif
                        //printf("Getting New A Block1\n ");fflush(stdout);
                        if(k+1 >= PRA && i+1 < PRB)//Bring first block in next column
                        {
                                TP=P;
                                TQ=Q;
                                // Mem copy host to device block a[k][i]
                                if(( (CA%P)>0) && (i+2 == PRB) )
                                {
                                        TP = (CA%P);
                                }
                                #ifdef PINNED
                                copyBlock(A , a, 0, i+1,TP,TQ, stream[1]);
                                #else
                                copyBlock(A , a, 0, i+1,TP,TQ);
                                #endif
                        }else if(k+1 < PRA && i+1 <= PRB)//Bring next block in column
                        {
                                TP=P;
                                TQ=Q;
```

```
                                if( i+1 == PRB && (CA%P)>0)
                                {
                                        TP=(CA%P);
                                }
                                if( k+2 == PRA && (RA%Q)>0)
                                {
                                        TQ=(RA%Q);
                                }

                                #ifdef PINNED
                                                copyBlock(A , a, k+1, i,TP,TQ, stream[1]);
                                #else
                                                copyBlock(A , a, k+1, i,TP,TQ);
                                #endif
                        }

                //printf("i=%d k=%d\n",i,k);
        }
        //printf("%d\n",i);
}//End main for loop

cublasShutdown();
cudaDeviceSynchronize();

if(TWork>=1)
{
        printf("Done GPU calculation...\n\nWaiting on %d threads in the host to
finish...\n\n",TWork);fflush(stdout);
}

while(TWork>=1)
{
        printf(".");
        usleep(1000);
}

gettimeofday (&TVCalcend, NULL);
differenceCalc = my_difftime (&TVCalcstart, &TVCalcend);

//Print C
printf("(.__.) <- Whale Done!!\n\nPrinting first 10 values of row in the middle:\n");fflush(stdout);
        for(int i=0; i<10; i++)
        {
                printf(" %.0f",C[PRC/2].block[i]);
        }
        printf("\n\nPrinting first value of every row:\n");
        for(int i=0; i<PRC; i++)
        {
                printf(" %.0f",C[i].block[0]);
```

```
        }
        printf("\n");
/*      printf("\n\nPrinting last value of every row:\n");
        for(int i=0; i<PRC; i++)
        {
                printf(" %.0f",C[i].block[(CB*Q)-1]);
        }
        printf("\n");

//For correctness purposes, please uncomment code below

   char* name="SaraMM_V1_Page_RES_6144.output";

          FILE * pFile;

          pFile = fopen (name,"w");
          TQ=Q;
          printf("First Q is=%ld",TQ);
           for(int i=0; i<PRC ;i++)
           {
                if(i+1 == PRC && PRC>1)
                {
                        TQ=LastTQ;
                        printf("Last Q is=%ld",TQ);
                }
                for (int n=0 ; n<CB*TQ ; n++)
                {
                        fprintf (pFile, "%.0f\n",C[i].block[n]);
                }
           }
          fclose (pFile);

          printf("\n\nPrinting %ld values TP= %ld TQ= %ld CB=%ld of C\n",TQ*CB,TP,TQ,CB);
          writeToFile(C[0].block,TQ*CB,"C_Sara_6144.output");
          printf("Printing %ld valuesof A\n",TP*TQ);
          writeToFile(A[0][0].block,TP*TQ,"A_Sara_6144.input");
          printf("Printing %ld valuesof B\n",TP*CB);
          writeToFile(Bb[0].block,TP*CB,"B_Sara_6144.input");
*/
          //For correctness purposes




printf("\nResult:\t\t%.0f\nShould be:\t%ld\n\n",C[PRC-1].block[0],CB);
//------------------------------------------------------------------------------
struct timeval TVEndstart, TVEndend;
gettimeofday (&TVEndstart, NULL);
//Cleaning up!
```

```
for(int si=0; si < SN ; si++)
{
	cudaFree(d_result[si]);
	cudaStreamDestroy(stream[si]);
}
for(int i=0;i<PRA;i++)
{
	free(C[i].block);
#ifdef PINNED
	cudaFreeHost(result0[i]);
#else
	free(result0[i]);
#endif


}

printf("Cleaning1\n ");fflush(stdout);

for(int i=0;i<PRB;i++)
{
#ifdef PINNED
	cudaFreeHost(Bb[i].block);
#endif
}
//printf("Cleaning2\n ");fflush(stdout);
cudaFree(a[0].block);
cudaFree(b[0].block);

gettimeofday (&TVEndend, NULL);
differenceEnd = my_difftime (&TVEndstart, &TVEndend);
printf("-----------------TIMING-------------------------\n");
printf ("Time of Init calculation:\t%3d.%2d secs.\n", difference->secs, difference->usecs);
printf ("Time of GPU calculation:\t%3d.%2d secs.\n", differenceCalc->secs, differenceCalc->usecs);
printf ("Time of Cleaning:\t\t%3d.%2d secs.\n", differenceEnd->secs, differenceEnd->usecs);
printf ( "\n\n Device to Host data MB= %.2f \n\n", totalSize);
free(difference);
free(differenceCalc);
free(differenceEnd);

//cudaDeviceReset();


#endif

	return 0;
}
```

```
int MMC_Fits() // Solver_1
{
	//Time variables
	TIME_DIFF * difference;
	TIME_DIFF * differenceCalc;
	TIME_DIFF * differenceEnd;
	struct timeval TVInitstart, TVInitend;
	gettimeofday (&TVInitstart, NULL);

	long long int allocSize = (long long int)(BC*B)*WC*W*sizeof(float);
	allocSize = allocSize / (1024*1024);
	SN=1; //Number of streams to use for the GPGPU execution
	//Allocation Intelligence
	int Maxw=1;

	printf("\n\n---Matrix Fit in 90Per GPU Memory---\nOptimization Enabled - The algorithm will try to use most of GPU Memory...\n");
	long long int memSize=(long long int) 5 * 1024 * 1024 * 1024;

	long long int currentElements= (long long int)(BC*WC)*(B*Maxw) * (SN+1);//Current Row Size in GPU
	long long int TotalElements= (long long int)(BC*B)*WC*W;		//Total Matrix C Elements
	long long int MaxElements= (long long int) (memSize/4);		//Maximum ELements we can have in GPU

	int T=1;
	long int RA=WC*BC;
	long int CA=W*B;
	long int RB=W*B;
	CB = WC*BC;					//Total elements in one dimension
	long long int Q= W * B;
	long long int Celemen=CB*Q;
	MaxElements -= Celemen;

	printf("\nCurrentElements=\t%lld\nTotalElements=\t\t%lld\nMaxElements=\t\t%lld\n\n",currentElements,TotalElements,MaxElements);

	long int EGMS =(long int)  MaxElements;
	long int P = (long int)((EGMS*1.0)/((Q+CB)));
	if( P > ceil((1.0*CB)/T))
	{
		P=ceil((1.0*CB)/T);
	}

	long int PCA = ceil((1.0*CA)/P);  //# Columns of Blocks of A - this could be interpreted as the number of columns to solve
```

```
	if(PCA>1)
	{
		P=(1.0*CA/PCA);
	}
	long int X = P;

	long int PRA = ceil((1.0*RA)/Q);  //# Rows of Blocks of A
	long int PRB = ceil((1.0*RB)/P);   //# Rows of Blocks of B
			PCA = ceil((1.0*CA)/P);   //# Columns of Blocks of A - this could be interpreted as the
number of columns to solve

	long int PRC = PRA;				//# Rows of Blocks of C

	long int AEGMS = (( Q*P + P * CB + SN*(Q*CB)) * sizeof(float)) /1024/1024; //Spaced used in GPU per
computation
	printf("\nCA=\t\t%ld\n"
			"RA=\t\t%ld\n"
			"PCA=\t\t%ld\n"
			"PRA=\t\t%ld\n"
			"CB=\t\t%ld\n"
			"RB=\t\t%ld\n"
			"PRB=\t\t%ld\n"
			"EGMS=\t%ld MB\n"
			"P=\t\t%ld\n"
			"X=\t\t%ld\n"
			"Q=\t\t%ld\n"
			"AEGMS=\t%ld MB\n"
			"\n",CA,RA,PCA,PRA,CB,RB,PRB,(EGMS*sizeof(float))/1024/1024,P,X,Q,AEGMS);

	BLOCKSIZE=(int)sizeof(float)*P*Q;
	long int TP=P;
	long int TQ=Q;

	//Change global values if possible
	//End allocation Intelligence

	//return 0;

	//cublasSgemm init Values -- A = m x k , B = k x n
	char transa='n';
	char transb='n';
	int m=BC;
	int n= BC * WC; //Size of B (in device) is a whole Row
	int kk=B;
	int lda=m;//m;		//Leading coordinate size for matrix A
	int ldb=kk;//kk;		//Leading coordinate size for matrix B
	int ldc=m;//m;		//Leading coordinate size for matrix C
	//end cublasInitValues
```

```
//Initialize Matrices
Matrix** A;
Matrix* Bb;
Matrix* C;
Matrix* b;//Sub row of blocks of Bb

#ifdef PINNED
		cudaError_t cError;
		cError = cudaHostAlloc( (void**) &A,sizeof(Matrix*) * PRA , cudaHostAllocDefault);
			if(cError!=cudaSuccess)
			  {
				printf("Error Allocating Pinned Memory Memory at A\n");
				return 0;
			  }
		cError = cudaHostAlloc( (void**) &Bb,sizeof(Matrix) * PRB , cudaHostAllocDefault);
			if(cError!=cudaSuccess)
			  {
				printf("Error Allocating Pinned Memory Memory at Bb\n");
				return 0;
			  }
				C = (Matrix*) malloc( sizeof(Matrix) * PRC );
		cError = cudaHostAlloc( (void**) &b,sizeof(Matrix), cudaHostAllocDefault);
			if(cError!=cudaSuccess)
			  {
				printf("Error Allocating Pinned Memory Memory at b\n");
				return 0;
			  }

		echo("Allocating Host Memory - Pinned");
		printf("%lld MBs per Matrix (A,B,C)-> ",allocSize);

		TP=P;
		TQ=Q;
		BLOCKSIZE=sizeof(float)*TP*TQ;
		for(int i=0; i< PRA ; i++)
		{
			if(i+1==PRA && ((RA%Q)>0) && PRA>1 )
			{
				BLOCKSIZE=sizeof(float)*(P)*(RA%Q);
				TQ=(RA%Q);
			}
			A[i]=new Matrix[PCA];
			for(int k=0; k<PCA;k++)
			{
				std::uninitialized_fill_n(A[i][k].block, TP*TQ, 1);
			}
		}
```

```
                TP=P;
                for(int i=0; i< PRB ; i++)
                {
                        if(i+1==PRB && (RB%P)>0 && PRB>1)
                        {
                                TP=(RB%P);
                                printf("Allocating last small row of B\n");fflush(stdout);
                        }
                        Bb[i].block=(float*) malloc( sizeof(float)*CB*TP );
                        std::uninitialized_fill_n(Bb[i].block, CB*TP, 1);

                }

#else

                A = (Matrix**) malloc( sizeof(Matrix*) * PRA );
                Bb = (Matrix*) malloc( sizeof(Matrix) * PRB );
                C = (Matrix*) malloc( sizeof(Matrix) * PRC );
                echo("Allocating Host Memory - Pageable");
                printf("%lld MBs per Matrix (A,B,C)-> ",allocSize);fflush(stdout);

        srand(2006);

        // initialize host memory

                TP=P;
                TQ=Q;
                BLOCKSIZE=sizeof(float)*TP*TQ;
                for(int i=0; i< PRA ; i++)
                {
                        if(i+1==PRA && ((RA%Q)>0) && PRA>1 )
                        {
                                BLOCKSIZE=sizeof(float)*(P)*(RA%Q);
                                TQ=(RA%Q);
                        }
                        A[i]=new Matrix[PCA];
                        for(int k=0; k<PCA;k++)
                        {
                                std::uninitialized_fill_n(A[i][k].block, TP*TQ, 1);
                        }
                }


                TP=P;
                for(int i=0; i< PRB ; i++)
                {
                        if(i+1==PRB && (RB%P)>0 && PRB>1)
                        {
                                TP=(RB%P);
```

```
                                printf("Allocating last small row of B\n");fflush(stdout);
                        }
                        Bb[i].block=(float*) malloc( sizeof(float)*CB*TP );
                        std::uninitialized_fill_n(Bb[i].block, CB*TP, 1);

                }

#endif

                b = (Matrix*) malloc (sizeof(Matrix));

                //For Correctness purposes, please comment code below

                //readAndAllocate(A[0][0].block,(TP * TQ), "../A.gold" );
                //readAndAllocate(Bb[0].block,(CB * TP), "../B.gold" );

                //For Correctness purposes, please comment code above

printf("Done!\nInitialization of Matrices-> ");fflush(stdout);

//matrixStrucInit(A,1);   //Initiate all values of the internal Matrix to the second parameter

BLOCKSIZE=(int) sizeof(float)*P*Q;
printf("Here\n");fflush(stdout);
// Initiate device containers
float* temp;
Matrix *a= new Matrix[1];         //Sub block of A
cudaMalloc((void**) &temp, sizeof(float)*P*Q);
a[0].block = temp;

// Allocate and initialize an array of stream handles for Asynchronous instruction issue.
cudaStream_t *stream = (cudaStream_t *) malloc(SN * sizeof(cudaStream_t));    //Kernel executers
cudaStreamCreate(&Dstream);// DtoH global stream
for (int i = 0; i < SN; i++)
{
   cudaStreamCreate(&(stream[i]));
}

float** d_result=(float**)malloc(sizeof(float*) * SN);      //Result container at Device, note there is one
result container per Stream

BLOCKSIZE=CB*Q;
for (int i = 0; i < SN; i++)
{
        cudaMalloc((void**) &d_result[i],CB*Q * sizeof(float));//Because A=m*k B=k*n C=m*n &&
m=WC*BC k=W*B n=WC*BC
        cudaMemset(d_result[i], 0, CB*Q * sizeof(float) );
}
```

```
	BLOCKSIZE=(int)CB*P*sizeof(float);
	cudaMalloc((void**) &temp,CB*P*sizeof(float));
	b[0].block = temp;

	TMat* container=(TMat*) malloc( sizeof(TMat)* PRA);   //Package sent to a thread for host addition.
	cudaEvent_t event[PRA];                                                    //Events to
synchronize kernel and DtoH memCopy
	pthread_attr_t attr;                                                       //Pthread attributes
	pthread_attr_init(&attr);
	pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
	#pragma omp parallel for schedule(dynamic)
	for(int e=0; e < PRA; e++)
	{
		cudaEventCreate (&event[e]);
	}

	gettimeofday (&TVInitend, NULL);
	difference = my_difftime (&TVInitstart, &TVInitend);

	struct timeval TVCalcstart, TVCalcend;
	gettimeofday (&TVCalcstart, NULL);

	TP=P;
	TQ=Q;

	printf("Start to solve the Matrix Multiply WC= %d W=%d BC=%d B=%d Matrix Size= %d x %d\n\nStart
GPU Calculation...\n",WC,W,BC,B, WC*BC , WC*BC);fflush(stdout);

	int sc=0;
	#ifdef PINNED
	copyBlock(A , a, 0, 0,TP,TQ, stream[0]);//Copy First block before we start
	#else
	copyBlock(A , a, 0, 0,TP,TQ);
	#endif
	cublasInit();

	for(int i=0; i < PRB ; i++)//Move per Row
	{
		if(i+1==PRB && (RB%P)>0 && PRB>1)
		{
			printf("Changing values of CUBLASSGEMM...\n ");fflush(stdout);
			TP=(RB%P);
			//-- A = m x k , B = k x n
			m=TQ;//Reminder of RA
			n= CB;   //Size of B (in device) is a whole Row
			kk=TP;//Reminder of RB
			lda=kk;//m;		//Leading coordinate size for matrix A
			ldb=n;//kk;		//Leading coordinate size for matrix B
			ldc=n;//m;		//Leading coordinate size for matrix C
```

```
                        //end cublasInitValues
                }else
                {
                        TP=P;
                        printf("Hereee...TP(kk)=%ld\tTQ(m)=%ld\tCB(n)=%ld\n ",TP,TQ,CB);fflush(stdout);
                        //-- A = m x k , B = k x n
                        m=(TQ);//Reminder of RA
                        n= CB;   //Size of B (in device) is a whole Row
                        kk=(TP);//Reminder of RB
                        lda=kk;//m;                 //Leading coordinate size for matrix A
                        ldb=n;//kk;                 //Leading coordinate size for matrix B
                        ldc=n;//m;                  //Leading coordinate size for matrix C
                        //end cublasInitValues
                }
//              cudaDeviceSynchronize();
                printf("\n\tRow [%d] of %ld being computed...\n\n",i+1,PRB);fflush(stdout);


#ifdef PINNED
                copyRow(Bb, b, i,TP, stream[0]);//Copy First block before we start
#else
                copyRow(Bb, b, i,TP);
#endif


                for(int k=0; k < PRA ; k++)//Move per Column - Per A block in a row
                {
                        sc=k%SN;//Decide which stream to use.
                        printf("RowA[%d] of %ld \n ",k+1,PRA);fflush(stdout);
                        long int TQ=Q;
                        BLOCKSIZE= sizeof(float)*(CB)*(Q);
                        if( k+1 == PRA && (RA%Q)>0)
                        {
                                BLOCKSIZE=sizeof(float)*(CB)*(RA%Q);
                                TQ=(RA%Q);
                        }
                        printf("C - TP= %ld TQ= %ld \n ",TP,TQ);fflush(stdout);
#ifdef PINNED
                        cublasSetKernelStream(stream[sc]);              //Tell device which stream to use for
cublasSgemm()
                        cublasSgemm(transa, transb, n, m, kk, 1.0f, b[0].block, ldb, a[0].block, lda, 1.0f,
d_result[sc], ldc);//SGemm
#else
                        cublasSgemm(transa, transb, n, m, kk, 1.0f, b[0].block, ldb, a[0].block, lda, 1.0f,
d_result[sc], ldc);//SGemm
#endif

                        printf("Done SGEMM Call\n");fflush(stdout);
                                if(k+1 >= PRA && i+1 < PRB)//Bring first block in next column
```

```
                        {
                                TP=P;
                                TQ=Q;
                                // Mem copy host to device block a[k][i]
                                if(( (CA%P)>0) && (i+2 == PRB) )
                                {
                                        TP = (CA%P);
                                }
                                #ifdef PINNED
                                copyBlock(A , a, 0, i+1,TP,TQ, stream[0]);
                                #else
                                copyBlock(A , a, 0, i+1,TP,TQ);
                                #endif
                        }else if(k+1 < PRA && i+1 <= PRB)//Bring next block in column
                        {
                                TP=P;
                                TQ=Q;

                                if( i+1 == PRB && (CA%P)>0)
                                {
                                        TP=(CA%P);
                                }
                                if( k+2 == PRA && (RA%Q)>0)
                                {
                                        TQ=(RA%Q);
                                }

                                #ifdef PINNED
                                                copyBlock(A , a, k+1, i,TP,TQ, stream[0]);
                                #else
                                                copyBlock(A , a, k+1, i,TP,TQ);
                                #endif
                        }


                //printf("i=%d k=%d\n",i,k);
        }
        //printf("%d\n",i);
}//End main for loop



BLOCKSIZE=CB*Q;
for(int i=0; i< PRC ; i++)
{
        if(i+1==PRC && (RA%Q)>0)
        {
                BLOCKSIZE=(int) CB*(RA%Q);
        }
```

```
		//C[i].block=(float*) calloc ( BLOCKSIZE ,sizeof(float));//Init of result matrix
		C[i].block=(float*) malloc( sizeof(float)*CB*Q );

	}
	cudaDeviceSynchronize();
	printf("Bringing results..\n\n");fflush(stdout);
	for (int i = 0; i < PRA; i++)
	{
		sc=i%SN;//Decide which stream to use.
		cudaMemcpy(C[i].block, d_result[sc],sizeof(float) * CB * Q , cudaMemcpyDeviceToHost);
	}

	cublasShutdown();

	if(TWork>=1)
	{
		printf("Done GPU calculation...\n\nWaiting on %d threads in the host to
finish...\n\n",TWork);fflush(stdout);
	}

	while(TWork>=1)
	{
		printf(".");
		usleep(1000);
	}
	gettimeofday (&TVCalcend, NULL);
	differenceCalc = my_difftime (&TVCalcstart, &TVCalcend);

	//Print C
	printf("(.__.) <- Whale Done!!\n\nPrinting first 10 values of row in the middle:\n");fflush(stdout);
	//sleep(5);
		for(int i=0; i<10; i++)
		{
			printf(" %.0f",C[PRC/2].block[i]);
		}
		printf("\n\nPrinting first value of every row:\n");
		for(int i=0; i<PRC; i++)
		{
			printf(" %.0f",C[i].block[0]);
		}
		printf("\n");
	/*	printf("\n\nPrinting last value of every row:\n");
		for(int i=0; i<PRC; i++)
		{
			printf(" %.0f",C[i].block[(CB*Q)-1]);
		}
		printf("\n");

	//For correctness purposes, please uncomment code below
```

```
   char* name="SaraMM_V1_Page_RES_6144.output";

            FILE * pFile;

            pFile = fopen (name,"w");
            TQ=Q;
            printf("First Q is=%ld",TQ);
             for(int i=0; i<PRC ;i++)
             {
                    if(i+1 == PRC && PRC>1)
                    {
                            TQ=LastTQ;
                            printf("Last Q is=%ld",TQ);
                    }
                    for (int n=0 ; n<CB*TQ ; n++)
                    {
                            fprintf (pFile, "%.0f\n",C[i].block[n]);
                    }
             }
            fclose (pFile);

            printf("\n\nPrinting %ld values TP= %ld TQ= %ld CB=%ld of C\n",TQ*CB,TP,TQ,CB);
            writeToFile(C[0].block,TQ*CB,"C_Sara_6144.output");
            printf("Printing %ld valuesof A\n",TP*TQ);
            writeToFile(A[0][0].block,TP*TQ,"A_Sara_6144.input");
            printf("Printing %ld valuesof B\n",TP*CB);
            writeToFile(Bb[0].block,TP*CB,"B_Sara_6144.input");
*/
            //For correctness purposes



printf("\nResult:\t\t%.0f\nShould be:\t%ld\n\n",C[PRC-1].block[0],CB);
//---------------------------------------------------------------------------------
struct timeval TVEndstart, TVEndend;
gettimeofday (&TVEndstart, NULL);
//Cleaning up!


for(int si=0; si < SN ; si++)
{
          cudaFree(d_result[si]);
          cudaStreamDestroy(stream[si]);
}
for(int i=0;i<PRA;i++)
{
          free(C[i].block);
```

```
	}

	printf("Cleaning1\n ");fflush(stdout);
#ifdef PINNED
	for(int i=0;i<PRB;i++)
	{
		cudaFreeHost(Bb[i].block);
	}

	for(int i=0; i< PRA ; i++)
	{
		for(int k=0; k<PCA;k++)
		{
			cudaFreeHost(A[i][k].block);
		}
	}
#endif
	//printf("Cleaning2\n ");fflush(stdout);
	cudaFree(a[0].block);
	cudaFree(b[0].block);

	gettimeofday (&TVEndend, NULL);
	differenceEnd = my_difftime (&TVEndstart, &TVEndend);
	printf("-----------------TIMING-------------------------\n");
	printf ("Time of Init calculation:\t%3d.%2d secs.\n", difference->secs, difference->usecs);
	printf ("Time of GPU calculation:\t%3d.%2d secs.\n", differenceCalc->secs, differenceCalc->usecs);
	printf ("Time of Cleaning:\t\t%3d.%2d secs.\n", differenceEnd->secs, differenceEnd->usecs);
	free(difference);
	free(differenceCalc);
	free(differenceEnd);
	cudaDeviceReset();

	return 0;
}



void randomInit(float *data, int size)
{
   for (int i = 0; i < size; ++i)
      data[i] = rand() / (float)RAND_MAX;
}

int writeToFile(float* array, long int elemNum,char* name)
{
	FILE * pFile;

	pFile = fopen (name,"w");
	for (int n=0 ; n<elemNum ; n++)
```

```
            {
              fprintf (pFile, "%.0f\n",array[n]);
            }
            fclose (pFile);

          return 0;
}

int readAndAllocate(float* container, int elemNum, char* name)
{

   float in;
   ifstream infile(name);

   if (!infile) {
      cout << "There was a problem opening file "
           << name
           << " for reading."
           << endl;
      return 1;
   }
   cout << "Opened " << name << " for reading." << endl;
   int count=0;

   for(int i=0; i<elemNum ; i++)
   {
          infile >> in;
          container[i]=in;
          count++;
   }
   /*
   while (infile >> in)
   {
          container[count]=in;
          count++;
   }
   */
   cout << "Read " << count << " floats and need to be " << elemNum << endl;

          return 0;
}


int AutoTune()
{
          //Matrix Size Variables
          SIZE=220;//800;//400;//200;//216//168//144//120//96//72//48//21//168//
          BSIZE=128;//32;//64;//128;
          ADJUSTC=1;
```

```
	ADJUST=1;

	int divisible=0;
	while(ADJUST<=(SIZE))
	{
		if(!(SIZE%ADJUST))
		{
			divisible++;
		}
		ADJUST++;
	}
	ADJUST=1;
	int* prospects=(int*)malloc(sizeof(int)*divisible);
	int i=0;
	while(ADJUST<=(SIZE))
	{
		if(!(SIZE%ADJUST))
		{
			prospects[i]=ADJUST;
			i++;
		}
		ADJUST++;
	}
	ADJUST=1;

for(int k=0;k<divisible;k++)
{
	printf(" %d",prospects[k]);
}
printf("\n\nTotal number of prospects=%d\n----------------------------------------\n",divisible);

float ave=0;
float bestTime[3];//Time,AdjustC,Adjust
bestTime[0]=1000000;
bestTime[1]=0;
bestTime[2]=0;

float aveX=0;
float bestX[3];//Time,AdjustC,Adjust
bestX[0]=1000000;
bestX[1]=0;
bestX[2]=0;


cudaEvent_t start, stop;
cudaEventCreateWithFlags(&start,cudaEventBlockingSync);
cudaEventCreateWithFlags(&stop,cudaEventBlockingSync);
int goodNums=0;
```

```
cublasInit();
//for(int k=divisible-1;k>=0;k--)
//{
	printf("----------------------------------------\n");
	//ADJUSTC=prospects[k];

	for(int p=0;p<divisible;p++)
	{
		ADJUST=prospects[p];
		ADJUSTC=prospects[p];
	//This code is square matrixes.
	BC=BSIZE*ADJUSTC;//128//96 //Device Block Size on the x direction (24 for this specific device C2075)
	WC=SIZE/ADJUSTC;//78//(MP*4)*2	//Width is the number of blocks per row which has to be W % 2
= 0
	B=BSIZE*ADJUST;//128//Device Block Size on the y direction (32 for this specific device C2075)
	W= SIZE/ADJUST;//78	//Width is the number of blocks per column which has to be W % 2 = 0
	BLOCKSIZE=(int) sizeof(float)*BC*B;

	float*aH = (float*) malloc(sizeof(float)* B*BC);
	//float*BbH= (float*) malloc(sizeof(float)* B*BC*WC);
	float*BbH= (float*) malloc(sizeof(float)* BC*BC*WC);
	float*CcH = (float*) malloc(sizeof(float)* B*BC*WC);
	float* aD;
	float* BbD;
	float* CcD;
	cudaMalloc((void**) &aD,BLOCKSIZE);
	cudaMalloc((void**) &BbD,BC*BC*sizeof(float)* WC);
	cudaMalloc((void**) &CcD,BLOCKSIZE * WC);

	for(int i=0;i<B*BC;i++)
	{
		aH[i]=1;
	}
	for(int i=0;i<B*BC*WC;i++)
	{
		CcH[i]=0;
	}
	for(int i=0;i<BC*BC*WC;i++)
	{
		BbH[i]=1;
	}

	//cudaMemcpy(aH, aD,BLOCKSIZE, cudaMemcpyHostToDevice);
	//cudaMemcpy(BbH, BbD,BLOCKSIZE * WC, cudaMemcpyHostToDevice);
	//cudaMemcpy(CcH, CcD,BLOCKSIZE * WC, cudaMemcpyHostToDevice);

	//cublasSgemm init Values -- A = m x k , B = k x n
	char transa='n';
	char transb='n';
```

```
	int m=B;
	int n= BC * WC; //Size of B (in device) is a whole Row
	int kk=BC;
	int lda=m;//m;		//Leading coordinate size for matrix A
	int ldb=kk;//kk;		//Leading coordinate size for matrix B
	int ldc=m;//m;		//Leading coordinate size for matrix C
	cublasInit();
	//end cublasInitValues


	long long int allocSize = (long long int)(BC*B)*WC*W*sizeof(float);
	allocSize = allocSize / (1024*1024);

	float time;

	cudaMemcpy(aH, aD,BLOCKSIZE, cudaMemcpyHostToDevice);
	//cudaMemcpy(BbH, BbD,BLOCKSIZE * WC, cudaMemcpyHostToDevice);
	cudaMemcpy(BbH, BbD,BC*BC * WC* sizeof(float), cudaMemcpyHostToDevice);
	cudaMemcpy(CcH, CcD,BLOCKSIZE * WC, cudaMemcpyHostToDevice);
	cudaEventRecord(start, 0);
	for(int q=0;q<3;q++)
	{
		cublasSgemm(transa, transb, m, n, kk, 1.0f, aD, lda, BbD, ldb, 0.0f, CcD, ldc);//SGemm
		/*if(status != CUBLAS_STATUS_SUCCESS)
		{
			printf("there was an error with code %d\n",status);
			break;
		}*/
	}
	cudaEventRecord(stop, 0);
	cudaEventSynchronize(stop);
	cudaMemcpy(CcH, CcD,BLOCKSIZE * WC, cudaMemcpyDeviceToHost);
	time=1;
	cudaEventElapsedTime(&time, start, stop);
	//cudaEventDestroy(start);
	//cudaEventDestroy(stop);
	time=time/3;
	//ave+=time;//((difference->secs>0)?difference->secs:difference->usecs);
	int totalBlocks=W*WC;
	float expectedTimesec=(totalBlocks*(time/1000));
	//aveX+=expectedTimesec;
	printf
("WC=\t\t%d\nW=\t\t%d\nBC=\t\t%d\nB=\t\t%d\nADJUSTC=\t%d\nADJUST=\t\t%d\nTimePerB:\t\t%f
ms\n",WC,W,BC,B,ADJUSTC,ADJUST, time);
	printf("TotalBlocks=\t%d\nEstimatedTime=\t%f sec.\n\n",totalBlocks,expectedTimesec);
	//cudaMemcpy(CcH, CcD,BLOCKSIZE * WC, cudaMemcpyDeviceToHost);

if(bestTime[0] > time && time>0.5)
{
```

```
		bestTime[0]=time;
		bestTime[1]=ADJUSTC;
		bestTime[2]=ADJUST;
		ave+=time;//((difference->secs>0)?difference->secs:difference->usecs);
}
if(bestX[0] > expectedTimesec && expectedTimesec>2 )
{
		goodNums++;
		bestX[0]=expectedTimesec;
		bestX[1]=ADJUSTC;
		bestX[2]=ADJUST;
		aveX+=expectedTimesec;
}


		//Cleaning
		free(aH);
		free(BbH);
		free(CcH);
		cudaFree(aD);
		cudaFree(BbD);
		cudaFree(CcD);
//sleep(2);
		}

		//sleep(2);

//}
cudaEventDestroy(start);
cudaEventDestroy(stop);
cublasShutdown();

ave=ave/goodNums;

printf("--------------------------------------------------------------------------------\n");
printf("BestTime=\t%fms\nAverage=\t%.3f\n",bestTime[0],ave);
printf("ADJUSTC=\t%.0f\nADJUST=\t\t%.0f\n\n",bestTime[1],bestTime[2]);

aveX=aveX/goodNums;

printf("BestExpected=\t%fsec\nAverageXsec=\t%.3f\n",bestX[0],aveX);
printf("ADJUSTC=\t%.0f\nADJUST=\t\t%.0f\n\n",bestX[1],bestX[2]);

return 0;

}

int main(int argc, char** argv)
{
```

```
	//Matrix Size Variables
	SIZE=360;//216//168//144//120//96//72//48//21//168//3495000 Maximum Size for this GPGPU
	BSIZE=128;
	//This code is for square matrixes.
	ADJUST=1;
	MP=14;		//GPGPU multiprocessors
	GS=1;	//Grid size of each dimension
	BC=BSIZE*ADJUST;//128//96 //Device Block Size on the x direction (24 for this specific device C2075)
	WC=SIZE/ADJUST;//78//(MP*4)*2		//Width is the number of blocks per row which has to be W % 2
= 0
	B=BSIZE*ADJUST;//128//Device Block Size on the y direction (32 for this specific device C2075)
	W= SIZE/ADJUST;//78	//Width is the number of blocks per column which has to be W % 2 = 0

	//Cheking if one matrix fits in GPU memory

	long long int memSize=(long long int) 5 * 1024 * 1024 * 1024;
	long long int MaxElements= (long long int) (memSize/4);//Maximum ELements we can have in GPU
	CB = WC*BC; //Total elements in one dimension
	long long int Q= W * B;
	long long int Celemen=CB*Q;
	long long int Size_thres= (long long int)((96*MaxElements)/100);
	// End

	TWork=0;
	struct timeval TVstart, TVend;
	gettimeofday (&TVstart, NULL);
	TIME_DIFF * difference;

	if (Celemen>Size_thres )// && false)
	{
		MM();		//If No full matrix fits in GPU memory
	}else
	{
		MMC_Fits();	//If full matrix fits in 90% of GPU memory
	}

	gettimeofday (&TVend, NULL);
	difference = my_difftime (&TVstart, &TVend);
	printf ("Total Time: \t\t\t%3d.%2d secs.\n", difference->secs, difference->usecs);
	printf("-----------------------------------------------\n");
	free (difference);

	return 0;
}
```

Listing D.2: AHPCRCMM.cu – Scalable Matrix-Matrix Multiplication

Vita

Enrique Portillo was born in El Paso, Texas, on November 29, 1985 at 1:11 a.m., the son of Rafael Portillo Nevarez and Maria Guadalupe Vasquez Cervantes. After completing his degree at Bachilleres-5 in 2003, he entered the University of Texas at El Paso, receiving the degree of Bachelors of Science in Computer Science in 2011. He entered the graduate program in the Department of Computer Science at the University of Texas at El Paso in August 2011, receiving a Master's of Science in December 2013. During his stay at the University, he worked as a graduate research assistant for the High Performance Systems (HiPerSys) group, where he started his research in high performance systems.

Permanent address: 1534 Upson Dr.
El Paso, Texas 79902

This thesis was typed by Enrique Portillo.