Performance Analysis of GPU Memory Architectures with Standard Matrix-Multiplication in OpenCL

University of Colorado Denver
Department of Computer Science and Engineering
CSCI-5593 Advanced Computer Architecture

Shane Transue
shane.transue@ucdenver.edu

Alejandro Alonso
alejandro.alonso@ucdenver.edu

Sukthana Pongma
sukthana.pongma@ucdenver.edu

Abstract *– With the development of the uniform processor designs implemented in modern graphics processing units (GPUs) released by NVIDIA and AMD, the expansive development of parallel languages and frameworks allows us to create high performance parallel applications that exploit the architectural characteristics of these devices. Specifically, we look at the modern GPU memory hierarchy, the development of parallel algorithms in OpenCL and how fast on-chip local memory can be targeted to significantly increase the performance of modern parallel applications. We develop a standard benchmark, standard matrix multiplication in OpenCL, to analyze the desired performance gains achieved by using this memory. We have found that through the analysis of the elapsed execution time of our benchmark, the utilization of local on-chip memory versus global memory utilization has provided ~90% performance increase.*

I. Introduction

Recent developments in GPU architectures have transformed how parallel applications are developed, and thus spurred the rapid growth of parallel languages such as CUDA [1] and the open standard parallel library OpenCL [2]. Specifically these developments revolve around the unification of the graphics pipeline. Traditionally, GPU devices relied on a fixed-function pipeline for processing graphics information (mainly vertex operations, triangle rasterization, etc.). GPU devices relied on dedicated processors for processing these different types of data. Additionally, the address modes in these devices are limited (also contain limited instruction sets) and the output communication paths are all limited. This is due to the tightly structured vertex/fragment program structure that these GPU devices were developed for. This limits these devices to mainly fulfilling this one purpose; however experimental general purpose processing techniques were developed for GPUs with this architecture but they still had to adhere to this limited level of flexibility. These *hacks* [3] utilized texture memory as a generic data storage mechanism and implemented graphics shaders to perform the mathematical operations in parallel, generally writing the output to another texture. This workaround complicates the entire process of utilizing the capabilities that were already present in the hardware. Developers realized that this problem could be effectively simplified. The solution to this problem was the unified processor pipeline. This new architecture technique, pioneered by both AMD [4] and NVIDIA [5][6], allows all individual compute units to access standardized memory locations and perform any of the available operations specified by the program running on the GPU. Therefore with this change, all compute units can be utilized to execute the desired operations in parallel - across all processors in the GPU device. This provided the break-through that general purpose GPU computation needed to become widely adopted into modern parallel application design. Based on this architectural development we can now target all available compute units on a GPU device and utilize the standardized global and shared on-chip local memory to develop high-performance parallel applications. These developments have spurred widespread adoption of GPGPU algorithmic design in several fields such as image processing, database acceleration, financial analysis, and high performance computing (HPC).

Building on these developments we propose the creation of a standard matrix multiplication benchmark that utilizes either global or local on-chip shared memory, and we explore the performance gains that are achieved by targeting different levels of this newly standardized GPU memory architecture. This benchmark is developed as an OpenCL application that provides accurate timing information about the total execution of the parallel matrix multiplication algorithm. The results of this benchmark are then analyzed across a variety of GPUs.

II. Modern GPU Memory Hierarchies

The transformation of modern GPU architectures implemented by AMD and NVIDIA has provided a standardized, addressable interface upon which general purpose computations can be performed. The typical characteristics of a modern GPU memory hierarchy are largely defined by the unified processor architectural design. Since each compute unit requires memory to perform any operation, they all have access to a global memory array. This is generally the largest portion of on-device memory that is accessible by all compute units. This memory makes up the highest level of the memory hierarchy (on-device) and is generally the slowest. Lower level memories (some still heavily influenced by GPU graphics applications) with

lower latencies include constant memory, texture memory, and high level cache (generally L2). At the lowest level we consider the memories local to each compute unit: the L1 cache and user controlled shared local memory. With the exception of the compute units registers, these memory locations have the lowest memory latencies in modern GPU devices. This generalized version of most modern GPU memory hierarchies is illustrated in Figure 1.

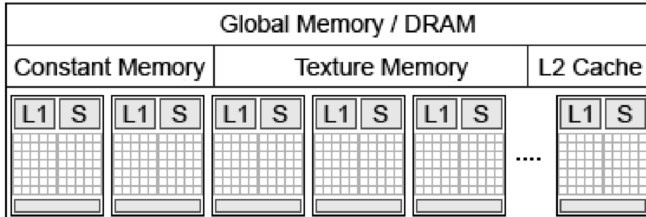


Figure 1. Generalized GPU Memory Hierarchy

To consider the performance of the memory hierarchy we wish to target the memory with the lowest latency as much as possible. Specifically we look at the memory with the lowest latency that can be algorithmically controlled: the on-chip shared local memory. This memory is local to a compute unit however it is shared among the processors within the compute unit and can be explicitly controlled at an application level. This is in contrast to typical application development where cache memory is not controlled at the application level, but rather implicitly accounted for. This is due to the extensive research into serial cache optimizations and recent developments in multi-core processors that automate the process of maintaining pertinent data elements for fast retrieval. This process has not been fully automated in parallel architectures with shared memory therefore it is explicitly addressable. Recent research [7] has attempted to automate this process by producing a polyhedral model of the source code in a parallel program and checking memory access patterns to optimize for shared memory. Yet this process cannot provide the full optimizations exhibited by each individual parallel application – in short, the process of fully optimizing shared memory use in parallel applications is still quite challenging and incomplete.

To ensure the accuracy of our implemented benchmark that we will use to analyze the difference between utilizing global memory and local memory, we will implement a well known standard parallel matrix multiplication algorithm. With this algorithm we can efficiently target the local on-chip memory provided by modern GPUs in a precisely controlled manner. The algorithm is easily decomposable into smaller sets of sub-problems that can be loaded into this local memory for an extremely efficient parallel algorithm implementation. This allows us to exploit the memory hierarchy of modern GPUs to illustrate the performance increase that shared memory utilization can provide.

Since we are interested in the performance increase of utilizing global versus shared local memory in all modern GPUs we must be able to execute the same parallel application on a variety of GPUs; this includes those that are from different vendors. Therefore to target all vendors we employ the OpenCL library to provide the implementation of our matrix multiplication benchmark.

III. OpenCL

OpenCL is the open standard for parallel programming of heterogeneous systems. This not only includes GPUs but also includes multi-core CPUs and other devices; however since we are targeting the memory hierarchy of modern GPUs, we will only be interested in running our OpenCL application on GPU devices. OpenCL provides a similar interface to devices that facilitate parallel computation as CUDA however there are some benefits and disadvantages to using this open standard. The purpose of OpenCL is to provide a standard interface to any of the parallel computational devices that are developed by any number of vendors. This standard is maintained by the Khronos Group [8] and allows independent vendors to implement the required interface with their own hardware specifications. Specifically we look at the vendor supplied implementations of the OpenCL interface by the AMD Accelerated Parallel Processing (APP) SDK and native driver support provided by NVIDIA.

This specification provides a layer of abstraction that allows OpenCL to target a wide range of devices. While this is a large benefit from an application standpoint, this generally means that the abstraction incurs performance penalties due to the fact that the code cannot be completely tailored to a specific device or architecture (this is something that CUDA is capable of since it’s maintained/developed by NVIDIA).

OpenCL is also different than CUDA in that OpenCL only provides a standardized C interface with is provided as a library. CUDA provides a compiler that is built upon C with additional syntactical structures for kernel specification and execution. Since the collaborative nature of OpenCL would make this approach infeasible, the library it provides is completely compatible with existing C/C++ compilers. This leads to yet another runtime penalty that OpenCL suffers from due to its abstraction layer. Since the specification of the kernel is not compiled at the time that the rest of the code is (as with CUDA) the application must compile the kernel code at runtime. The performance penalty incurred by this compilation is not extensive yet it will provide a much longer runtime than an equivalent CUDA application.

With these performance considerations we continue to utilize OpenCL for our benchmark application due to the fact that it is the *de facto* standard interface that can target all modern GPUs. This is simply because CUDA is developed by NVIDIA and targets their GPU architectures (Fermi, Kepler, etc.), and AMD has begun to support OpenCL as its main parallel computing environment interface. This allows us to run our benchmark application on all available popular

available GPU devices (while we are not directly comparing AMD and NVIDIA hardware, we are interested in the relative increase in performance that each vendor achieves through the utilization of shared on-chip memory instead of using naïve accesses to global memory).

IV. OpenCL Development

Targeting specific architectural elements cannot be achieved with OpenCL for the most part; one notable exception to this is the ability to specifically target shared local memory at an application (kernel) level. OpenCL and its parallel kernel language provide special variable modifiers that allow for user defined allocations on this on-chip memory reserved for each compute unit. Utilizing this ability we will develop two GPU kernel programs that perform matrix multiplication with (1) utilizing global memory accesses and (2) using local memory to minimize high latency global memory accesses. In order to achieve this we look at the OpenCL development model and how kernels are implemented and executed.

The OpenCL development model revolves around the abstract architecture that OpenCL defines for all compute devices in a system. Compute devices include but are not limited to GPUs and CPUs and device resides within the host (one of the available compute devices may even be the CPU of the host). In a typical desktop/laptop computer, a host may have access to two compute devices: generally a multi-core CPU and dedicated GPU. This architecture is illustrated in Figure 2.

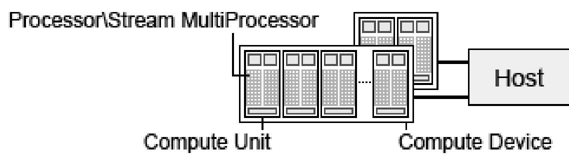


Figure 2. Abstract OpenCL Architecture

The two compute units in Figure 2. illustrate those typically found in a desktop/laptop computer where the multi-core CPU may have two or more compute units and the dedicated GPU may have several (typically 6 or more).

Compute devices are vendor specific, identified as platforms by OpenCL, and depending on which platform is selected, different devices will be available. Major vendors including those for CPU devices are Intel, AMD, and NVIDIA. For the OpenCL library to detect a device present in the system, the vendor of the device must provide the appropriate OpenCL enabling drivers. If these are not provided then OpenCL will fail to detect the required platform which is used to access the device. Figure 3. illustrates how one or more compute devices belong to a specified platform all within an OpenCL context (our benchmark implementation provides command-line arguments for selecting the desired platform and device based on what is automatically detected in the system). As per one of our configurations for example, two platforms are defined: (1) the AMD platform provided for the AMD Athlon 64 X2 6000+ processor and (2) the NVIDIA platform provided for the NVIDIA GeForce 260 GTX GPU.

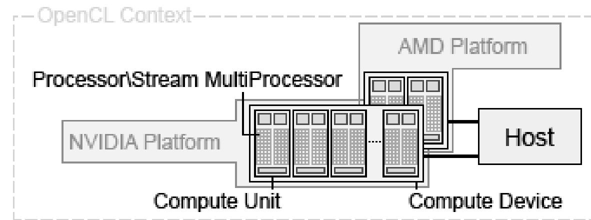


Figure 3. OpenCL Platforms and Associated Devices

With the selection of the desired platform and device, OpenCL memory operations and kernels can now be executed. In OpenCL these operations are enqueued to a specific device via a command queue. This provides a simple interface for performing memory operations and kernel executions on a specified device. This principle can be extended to each of the devices available in the system; however our simplistic benchmark application will only be taking advantage of a single device. Looking towards the development of our benchmark we utilize an active OpenCL context with a selected platform and device as illustrated above to perform our memory operations and kernel execution.

V. OpenCL Matrix Multiplication

The mathematical aspect of this computation is fairly straight forward. We take two square matrices and multiply them to get the result. Specifically we define our algorithm as a matrix multiplication between square matrices A and B of size ($mSize$ x $mSize$) to retrieve a resulting matrix C of size ($mSize$ x $mSize$) where the $*$ operator signifies matrix multiplication and $mSize$ is the dimension of one side of the square matrices:

$$C = A * B$$

This result is calculated in parallel through the execution of an OpenCL kernel and the matrix multiplication result is stored in the result matrix C when the computation is complete. To correspond to our simple definition of matrix multiplication, we have defined the pseudo-signature of our OpenCL kernels to match:

```
__kernel void matrixMultiply(A, B, C, mSize)
```

This signature is used for both versions of our OpenCL kernels. The first of which will utilize the high latency global memory of the GPU device and the second will use the low latency local memory available to each compute unit. These approaches have been well studied and thus the implementation of each is fairly straight-forward; however there are some OpenCL specifics that must be taken into

account when handling memory synchronization events which are encountered in the shared memory version of the kernel. The next two sections provide detailed explanations of each matrix multiplication algorithm implemented in our benchmark application using OpenCL.

VI. Matrix Multiplication in Global Memory

Matrix multiplication that utilizes global memory is effectively one of the easiest parallel algorithms to implement. The read-only independent nature of each resultant element lends this problem to natural fit in a parallel algorithm. Looking at the naïve serial matrix multiplication code we see that it can easily be converted to the parallel environment established by OpenCL (assuming n is defined as the matrix size $mSize$):

```
for ( int i = 0; i < n; i++ )  i=get_global_id(0);
 for ( int j = 0; j < n; j++ ) j=get_global_id(1);
  for ( int k = 0; k < n; k++ )
    C[i*n+j] += A[i*n+k] * B[k*n+j];
```

Here we understand that matrix multiplication is performed in two dimensions. Therefore we utilize a two dimensional kernel to effectively replace the two outer loops of the serialized algorithm. Therefore all corresponding loop iterations are executed in parallel. This provides the foundation of our matrix multiplication kernel that utilizes global memory. The next definitions to consider are the global work-size and work-group size. The global size is defined as the number of independent elements that will be executed in parallel from the problem definition. In the case of matrix multiplication this is simple: it is the size of the matrices. Since they are square we simply define the global work-size as a two dimensional problem: ($mSize$ x $mSize$). In OpenCL this is defined when the kernel is enqueued for execution and it is critical that this size is divisible by the work-group size, thus the next largest multiple of the work-group size defines the actual global work-size: ($mSize + x$, $mSize + x$), where x specifies the amount of additional work items that must be added to the global work-size to make it evenly divisible by the work-group size. This is illustrated in Figure 4.

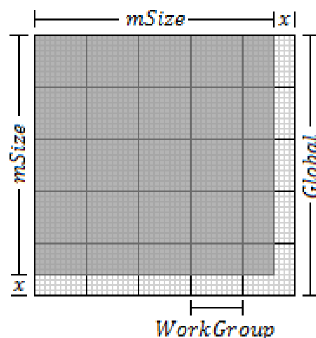


Figure 4. Global, work-group, and matrix size as they are enqueued as a kernel with a two dimensional range in OpenCL

The illustration in Figure 4. depicts that the global work-size is evenly divisible by the work-group size as required and is large enough to hold the dimensions of each matrix. The last requirement of this kernel is to properly check each work-items index to ensure that it is operating on a valid element within the boundaries of the actual matrix size. As illustrated in Figure 4, any element within range x should not be executed on any element from matrices A, B, or C. For the full source and additional documentation see listing: *opencl-mm-gpu1.cl*.

VII. Matrix Multiplication in Shared Memory

The matrix multiplication algorithm that utilizes shared memory is slightly more complex than the global memory algorithm proposed in the last section. However we can outline the process as follows: get the current work-group and work-item position, copy a block of memory from each input matrix to local memory, solve a sub-section of the global problem and write the result to the result matrix C. This version is inherently made more complex by the subdivision of the global problem to smaller localized blocks. When performing the algorithm in relation to the blocks the divide the global problem the indexing scheme changes. The following functions allow for the current block and thread indices to retrieved for working within a single block:

```
int tx = get_local_id(0):
int ty = get_local_id(1):

int c = get_group_id(0) * BLOCK_SIZE + tx;
int r = get_group_id(1) * BLOCK_SIZE + ty;
```

These functions allow us to define the limitations on when items should be accessed (eliminating out of bounds conditions as seen in Figure 4) and also allow us to convert the one dimensional array indices (into the globally allocated matrices A and B) into a set of two dimensional indices for copying the elements in global memory to local memory. The following lines define these shared local memory arrays in OpenCL and copy the contents from global memory to these locally allocated two dimensional arrays (if the indices are out of the bounds of the globally allocated matrices then the local arrays elements are set to 0.0).

```
__local float aLocal[BLOCK_SIZE][BLOCK_SIZE];
__local float bLocal[BLOCK_SIZE][BLOCK_SIZE];

if ( Row >= mSize && Col >= mSize ) {
  aLocal[ty][tx] = 0.0f;
  bLocal[ty][tx] = 0.0f;
}
else {
  aLocal[ty][tx] = A[ mSize * Row + (i + tx)];
  bLocal[ty][tx] = B[ mSize * (i + ty) + Col];
}
```

After loading the matrix elements from global memory to local memory we perform as many operations as possible

before storing the result and moving on to the next work-group (the rest of the matrix multiplication algorithm is performed in the same way as the version that utilizes global memory) This enables us extensively use this low latency memory to perform many of the required operations. The time saved during each access to this faster memory is where we derive our increased performance from this approach.

In this version we must consider the implications of the OpenCL memory synchronization API requirements. These requirements differ from the syncthreads() command typically found in CUDA applications. For memory synchronization in OpenCL the barrier(..) command is provided. The key difference between these synchronization functions is that the barrier(..) function requires that all threads in the current work-group must be able to execute the function. If a conditional branch is provided in the code it must not contain a barrier(..) call. This is because the function explicitly requires that "*If a barrier is inside a conditional statement, then all work-items must enter the conditional if any work-item enters the conditional statement and executes the barrier.*" [9]. This requirement is fulfilled by restructuring the conditional statements defined in this version of the algorithm. For the complete source code and extensive documentation on how this algorithm is performed and how the barrier(..) requirement is satisfied see listing: *opencl-mm-gpu2.cl*.

VIII. GPU Performance Analysis

The importance of GPUs and their extensive list of practical applications have created the necessity of studying their performance. In recent years there have been several studies about GPU performance modeling with some based in algorithmic execution time and others based on specific internal execution times. Zhang and Owens [10] proposed a NVIDIA exclusive CUDA based micro benchmark approach that allows for performance modeling at the instruction-pipeline level and analyzes shared/global memory access time in the hope that the program could be reviewed and improved once these bottlenecks are defined. Jurečko et al. [11] propose a vendor independent OpenCL based approach that tests the GPU operations such as sin, exp, log, addition, etc. These are typical arithmetic operations that are extensively used in GPGPU applications. In this researched approach kernels received the same random input data with maximum possible length several times and the numerical precision was measured from the outcome and finally the performance was measured as a throughput of its kernel divided by the average kernel runtime. Solano-Quinde et al. [12] analyze their unstructured grid analysis algorithm and GPU performance based on hardware occupancy and memory access efficiency to develop their implementation and model its performance based on its execution time.

Based on the development of our benchmark application we are attempting to measure the performance of two matrix multiplication algorithms. We wish to illustrate the difference between utilizing global and local shared memory on a variety of platforms with different GPU devices. We aim for a similar approach but expand our measurements to include GPU device memory allocation, kernel execution time and the GPU device to host result memory transfer. In doing this we also able to generate benchmarks relating the performance of our OpenCL-based matrix multiplication algorithm to an equivalent CUDA based algorithm to determine the relative speed between the two development environments in machines that contain NVIDIA hardware. Below is an exact breakdown of the events that are included in the calculation of the execution time used for our results:

(1) Buffer allocation on OpenCL device
(2) Kernel enqueued to OpenCL device and executed
(3) Read-back of result buffer to host memory

After (3) our benchmark program has the ability to print the entire result matrix C to a file, thus ensuring a correct result. The resulting execution time is then reported utilizing highly accurate platform-specific time measurement functions. However we do note the pitfall of running the benchmarks in this manner due to the external influences on the recorded performance, namely, the speed of main memory, the bandwidth to the GPU device and the speed of the host processor; all of which will impact the performance. We mitigate this problem by comparing machines with similar hardware specifications; while they are not completely equivalent (as is the case with laptops) they are within reasonable spec and will not excessively alter the asymptotic runtime analysis of the implemented algorithms.

IX. Benchmark Application

The development of our benchmark application is facilitated through the use of OpenCL and the C++ binding specification [13] provided by the Khronos group (there very few differences between using the C standard functions or the C++ bindings). With the library we provide a host-based command-line application that performs a number of parallel matrix-multiplication algorithms. Below is the list of matrix multiplication algorithms that are possible with the provided benchmark application:

(1) Global GPU Matrix Multiplication
(2) Shared GPU Matrix Multiplication
(3) Parallel CPU Matrix Multiplication

The implementation for each of these algorithms is provided through the use of OpenCL and the two kernels we have developed for performance measurement. Each of these algorithms can be selected from the command-line with a set of Boolean flags.

Something that had to be considered with this command-line based application was the selection of the appropriate OpenCL platform and device. Since a machine can have multiple platforms and devices we implemented a simple index selection system that allows the user to select,

by index, the appropriate platform and device they wish to test. Since this requires knowledge about the platforms and devices that OpenCL has detected we have provide a printout of all detected platforms and devices when the application is executed. From this point the user may look at the appropriate indices of the platform and device they wish to test and provide those as command-line arguments. The following provides an overview of the command-line arguments that are supported by our application (they must be precisely provided in the required order).

```
[0] <usage> Displays command-line information
    about the application
[1] <mult_version> Allows the user to select
    between global and shared matrix
    multiplication. Provide 1 for global and 2
    for shared.
[2] <mat_size> Defines the size of one
    dimension of the square matrices that will be
    used to perform the multiplication. If 10 is
    provided then the resulting matrix size will
    be (10 x 10)
[3] <platform_index> Allows the user to select the
    platform that contains the device they wish to
    test. Valid indices are 0 – (numPlatforms – 1)
    (this varies by system).
[4] <device_index> Allows the user to select the
    device they wish to run the application on.
    Depending on which platform is selected, the
    list of available devices will vary. Valid
    indices are (0 – numDevices – 1).
[5] <cpu> This flag tells the benchmark to run on
    the CPU using the parallel OpenCL kernel.
    Provide 0 to not use the CPU, 1 to use it.
[6] <print> If this flag is set to true then the
    resulting matrix C will be printed to a file
    (Note* that this creates an extremely large
    text file for large matrix sizes).
```

With these command-line arguments the benchmark application is able to provide an extensive amount of information about the execution time of the provided algorithms on a variety of devices. When the desired parameters are provided and the application is executed, it will return the total execution time detailed in the last section. Utilizing this application we have affirmed the performance increases that are provided by utilizing shared memory in modern GPU hierarchies across a variety of system configurations. The next section provides a detailed look at the tests conducted during this research.

X. Performance Benchmarking

To determine the relative performance increases achieved by utilizing shared memory over global memory in our OpenCL matrix multiplication benchmark application we conducted a series of tests on a wide variety of GPUs. While the systems containing the GPUs are quite different, we are only interested in the relative performance increase that each individual machine is capable of. With this information we can determine a number of things: How much of decrease in execution time can we achieve from utilizing shared memory? What GPU vendors see a larger benefit from utilizing shared memory? How does block size affect the efficiency of shared memory? How differently do OpenCL and CUDA applications perform on the same system (assuming NVIDIA hardware). We aim to answer these questions using the following list of available GPUs:

(1) NVIDIA GeForce GTX 260
(2) NVIDIA GeForce GTX 560
(3) AMD Radeon HD-6630M*
(4) NVIDIA GeForce 410M
(5) NVIDIA GeForce NVS 5200(M)

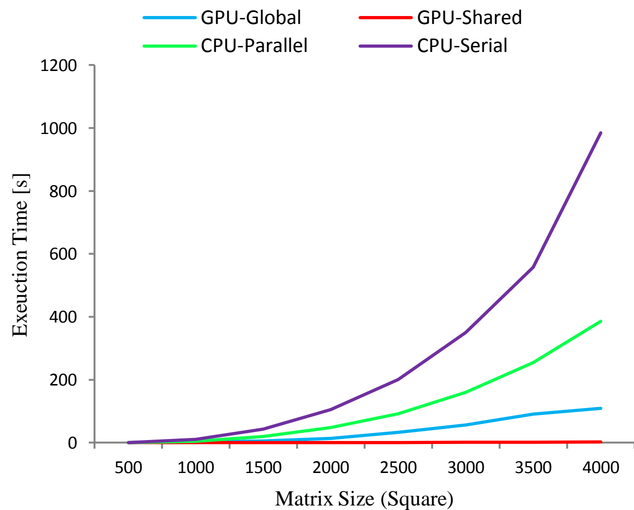
* The number of available AMD GPUs was limited at this time; however the HD-6630M and NVS-5200 are two competing GPUs in the same generation.

Through our extensive testing we provide insight into these questions and provide a glimpse into cross-vendor GPU analysis.

XI. Performance Analysis and Results

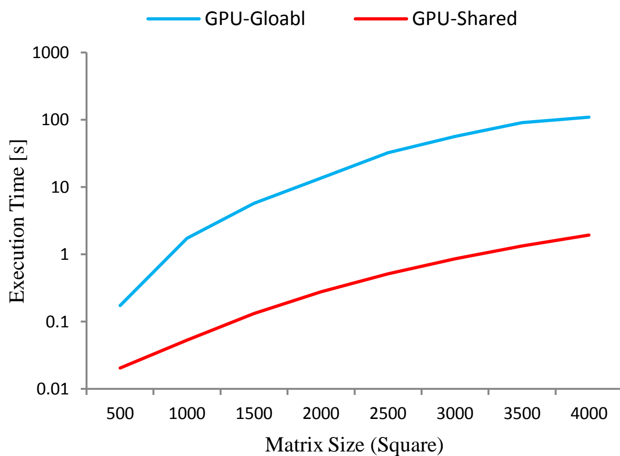
To summarize the intent behind this research we provide an overview of several benchmarks that illustrate the importance of effectively utilizing parallel computation and efficient use of low latency memory. The following set of benchmarks in Figure 5. provides a broad overview of how system architecture can drastically influence the required execution time of even a simple algorithm like matrix multiplication.

Matrix Multiplication Algorithmic Variants
(NVIDIA GeForce GTX 260)



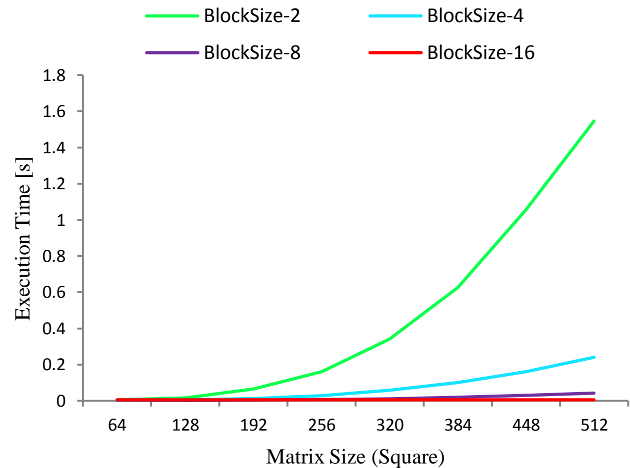
The Matrix Multiplication Algorithm Variants plot illustrates the drastic execution time differences between the several different potential matrix multiplication algorithms. All parallel implementations are provided through the OpenCL library as kernels. Unsurprisingly, the sequential algorithm has the worst runtime performance, yet even with a limited number of execution units the parallel CPU

algorithm provides a drastic decrease in execution time over this sequential algorithm. It is noticeable however that when we provide the same parallel algorithm to the massive number of parallel compute units found in a GPU that the performance is dramatically increased. While this is expected, we will further analyze the impact the GPU memory hierarchy plays in the large difference between the GPU-Global and GPU-Shared algorithms that exhibit the lowest runtimes. Analyzing this result on a logarithmic plot we can see the consistent increase in performance by the GPU algorithm that utilizes shared memory.

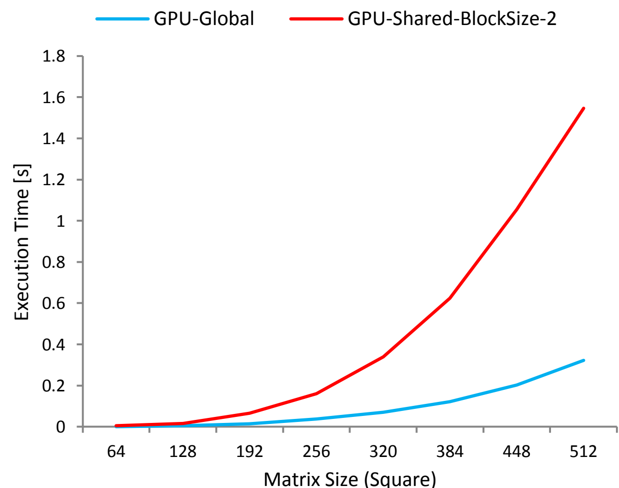


As expected, our experimental results consistently identify a large performance increase from the utilization of shared on-chip memory. The extensive use of the low latency memory drastically reduces the number of necessary memory accesses, however since the retrieval time does not suffer from the larger latency associated with global memory, the overall execution time is reduced. Over the entire set of provided matrix sizes tested we see an average 96.80% decrease in execution time. This is an incredible achievement; however we also consider the constraints on applying this approach to general problems. The structure of the matrix multiplication algorithm makes the process of utilizing shared memory to this extent extremely easy. Problems that exhibit more complex behavioral patterns and dependencies are not as well suited for this optimization. This is an important caveat that must be considered when analyzing these results. Yet we have clearly demonstrated that when the opportunity is available, shared on-chip memory should be utilized as often as possible to reduce overall execution time.

While utilizing shared memory is generally a good idea, other considerations must be made with respect to how efficiently the shared memory is used. Since the key aspect in designing an algorithm that utilizes shared memory is the ability to break down the global problem into smaller sub-problems, the question of how small each sub-problem should be. This is represented by our work-group size which in terms of our matrix multiplication algorithm, the block size used to divide the input matrices (according to Figure 4). In our performance analysis we illustrate that altering the block size used in the shared memory algorithm we can drastically alter the performance.



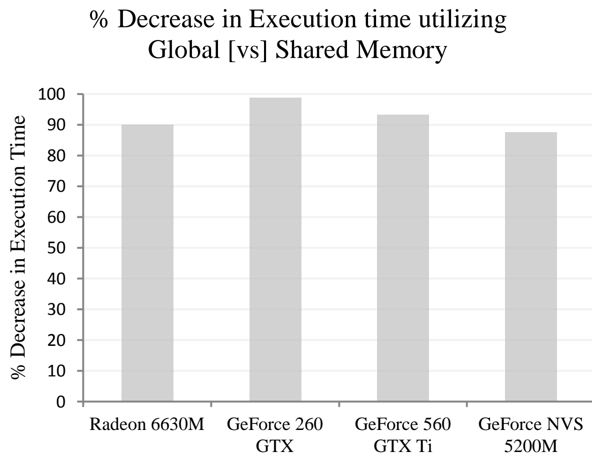
The results displayed in the plot above illustrate the negative impact of selecting an inappropriate block size for the execution of our algorithm. This reduction in performance can be explained by the inefficient utilization of the GPU execution pipeline. As the block size decreases the algorithm begins to approximate a sequential implementation. This is because the access to the shared memory for each work-group is serialized. Therefore this presents a form of thrashing – data from global memory must be constantly reloaded to shared memory for each work-group.



This produces an overhead that is so inefficient it actually has a longer execution time than the matrix multiplication algorithm utilizing global memory. This is illustrated in the previous plot. When utilizing shared memory it is critical to utilize the execution pipeline of the GPU, otherwise the performance benefits will be effectively eliminated. For our performance tests we conformed to the largest block size that was available to all GPUs within our test group: 16.

With these preliminary results we consider the relative performance increases demonstrated by our wide selection of modern GPUs. The plot below illustrates that across this set of GPUs we see an approximate 90% reduction in execution time by utilizing shared memory in our matrix multiplication algorithm.

% Decrease in Execution time utilizing
Global [vs] Shared Memory



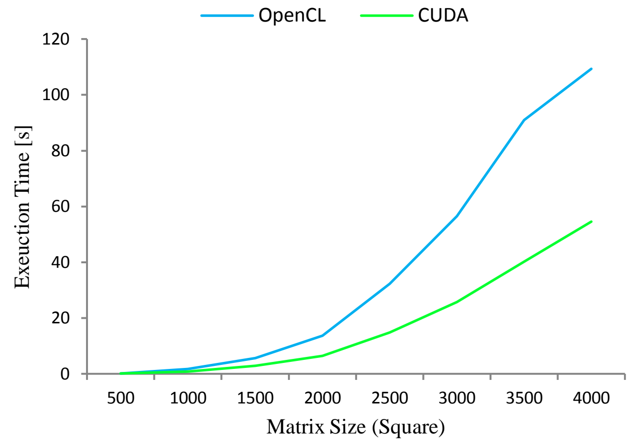
It is interesting to note here that there is no significant difference between GPU vendors with respect to shared memory performance. While this dataset is limited we can generally conclude that by utilizing shared memory in both AMD and NVIDIA hardware, the execution time can be decreased by ~90%. As per our objective we have effectively constructed and recorded the impressive reduction in execution time of the standard matrix by utilizing shared memory in our OpenCL multiplication algorithm.

XII. Extended Perormance Analysis

Our selection of modern GPUs includes several NVIDIA based devices which allows us to extend our analysis to compare the effective runtimes of our algorithms with equivalent CUDA implementations. Since the NVIDIA hardware supports both CUDA and OpenCL kernel execution we have extended our results to include this comparison. The theoretical answer to which of these parallel development paradigms may seem straight forward: we would expect that CUDA applications exploit all possible hardware optimizations to exhibit a lower execution time; while OpenCL applications lack this ability due to the heterogeneous design of OpenCL which forces abstraction of hardware specific details.

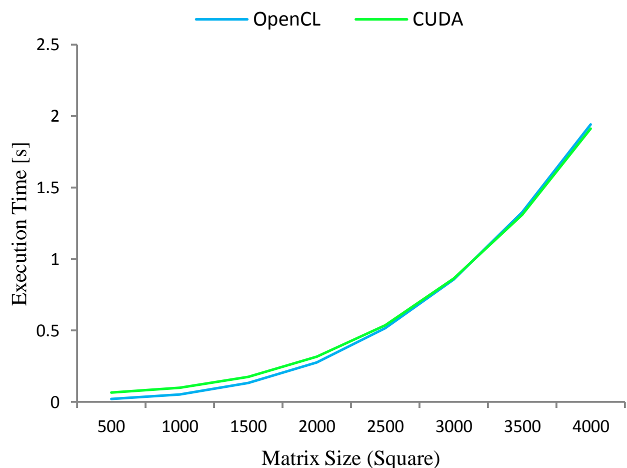
Provided with equivalent matrix multiplication kernels (containing a very similar number of total operations) developed in CUDA, all versions of the matrix multiplication algorithm were executed on the single system that contains a NVIDIA GTX 260.

Matrix Multiplication – OpenCL [vs] CUDA (Global)
(NVIDIA GeForce GTX 260)



We observe that as the matrix size increases the CUDA kernel implementation provides a lower execution time. This is interesting due to the fact that at the lowest level the generated code is executed on the same device. Additionally we consider if this trend will continue with the matrix multiplication algorithm that utilizes shared memory.

Matrix Multiplication – OpenCL [vs] CUDA (Shared)
(NVIDIA GeForce GTX 260)



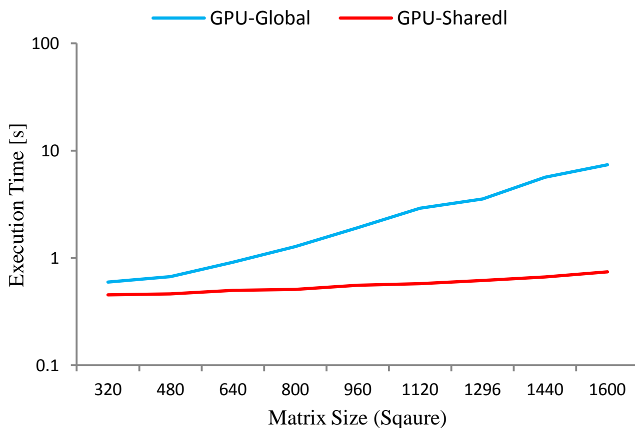
This presents a result that is initially somewhat perplexing. The matrix multiplication algorithm that targets shared memory essentially has an equivalent runtime for both OpenCL and CUDA version. The difference between these two results however can be further analyzed to understand how the memory architecture of the GPU is influencing this result. Initially consider the matrix

multiplication algorithm that utilizes shared memory. Since we are explicitly controlling how the low latency memory is used, we can accurately dictate the operations that are performed by the GPU. We effectively circumvent the additional implications of the devices cache simply because we explicitly control where each data block is located for the execution of each work-group. This is in contrast to the algorithms that utilize global memory. Since we are not explicitly in control of where the data elements are stored, the compiler and drivers can make form of appropriate decision on where and when to store the required data. Therefore we see that the CUDA implementation of the algorithm that utilizes global memory performs additional optimizations to reduce the overall execution time.

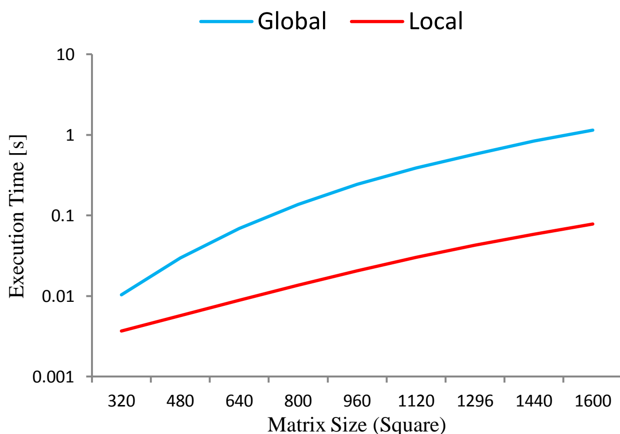
XIII. Miscalaneous Performance Results

During the course of our research and project implementation we performed several additional benchmarks on the GPUs listed in section X. These results are provided here.

Global [vs] Shared OpenCL Matrix Multiplication (Log)
(AMD Radeon HD 6630M)



Global [vs] Shared OpenCL Matrix Multiplication (Log)
(NVIDIA GeForce 560 GTX)



XIII. Conclusion

The aim of this research was to analyze the impact of modern GPU memory hierarchies on the performance of the standard dense matrix multiplication algorithm. We utilized this simple problem as a model for how shared memory can be extensively utilized to increase the performance of kernels implemented in OpenCL. Our results are consistent with this premise and effectively illustrate this point. We also conclude that when the matrix multiplication algorithm is implemented in shared on-chip memory we see a 90% performance increase over the equivalent matrix multiplication algorithm that utilizes global memory.

XIV. Future Work

At the time of executing these experimental results the support of monitoring executing OpenCL kernels in the NVIDIA visual profiler was somewhat inconsistent and therefore we elected to omit those results for consistency. Additionally the CodeXL package recently released still requires additional time to mature on a technical level Attempts at using this package were unsuccessful; however once the project is refined it will provide an excellent development tool for parallel application development.

The simplicity of the matrix multiplication problem and how well it can be subdivided into smaller problems presents a contrived result; most computationally expensive algorithms are not easily decomposable; hence why completely automated shared memory optimizers do not currently exist. To really appreciate the complexity involved in creating shared memory optimizations for general parallel applications a more complex case study is required.

The development of the benchmark application was successful however it would be nice to utilize additional libraries to closely watch the execution of the kernels and the utilization of the GPUs memory hierarchy. This would provide much more detailed information and a lot less rigorous investigation. The closest solution to this is to use the visual profilers under development by AMD and NVIDIA.

XV. References

[1] Compute Unified Device Architecture (CUDA) http://www.nvidia.com/object/cuda_home_new.html

[2] OpenCL – The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/

[3] NVIDIA Advanced CUDA Webinar – Memory Optimizations. http://on-demand.gputechconf.com/gtc-express/2011/presentations/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf

[4] AMD Graphics Core Next. http://www.amd.com/us/products/technologies/gcn/Pages/gcn-architecture.aspx

[5] NVIDIA's Next Generation CUDA Compute Architecture: Fermi http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf

[6] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[7] Junfeng Zhu; Gang Chen; Baifeng Wu; , "GPGPU Memory Estimation and Optimization Targeting OpenCL Architecture," *Cluster Computing (CLUSTER), 2012 IEEE International Conference on* , vol., no., pp.449-458, 24-28 Sept. 2012. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6337808&isnumber=6337766

[8] Khronos Group – Open Standards Maintainers. http://www.khronos.org/

[9] OpenCL API Specification. http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/barrier.html

[10] Yao Zhang; Owens, J.D., "A quantitative performance analysis model for GPU architectures," *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on , vol., no., pp.382,393, 12-16 Feb. 2011.*

[11] Jureko . Ko i ova . a . Kasanick , T.; Domiter, M.; Zvada, M., "Evaluation Framework for GPU Performance Based on OpenCL Standard," *Networking and Computing (ICNC), 2010 First International Conference on , vol., no., pp.256,261, 17-19 Nov. 2010.*

[12] Solano-Quinde L.; Wang Z.; Bode B.; Somani A., "Unstructured Grid Applications on GPU: Performance Analysis and Improvement", *GPGPU-4 Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, article 13*, 5 Mar. 2011.

[13] The OpenCL C++ Wrapper API. http://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf

XVI. Listings

opencl-mm-gpu1.cl – Provides the implementation of the OpenCL kernel that performs a matrix multiplication operation between matrices A and B utilizing shared memory and the stores the result in to matrix C.

```
__kernel void matrixMultiply(__global float* A,
                             __global float* B,
                             __global float* C,
                             int mSize) {

  int i = get_global_id(0);
  int j = get_global_id(1);

  float elementA = 0.0;
  float elementB = 0.0;

  if ( i < mSize && j < mSize ) {
      for (int k = 0; k < mSize; k++) {
      elementA = A[i * mSize + k];
      elementB = B[k * mSize + j];

      C[i * mSize + j] += elementA * elementB;
    }
  }
}
```

opencl-mm-gpu2.cl – Provides the implementation of the OpenCL kernel that performs a matrix multiplication operation between matrices A and B utilizing shared memory with a block size of 16 and then stores the result into matrix C.

```
#define BLOCK_SIZE 16
__kernel void matrixMultiply(__global float* A,
                             __global float* B,
                             __global float* C,
                             int mSize) {
  int tx = get_local_id(0);
  int ty = get_local_id(1);

  int Row = get_group_id(1) * BLOCK_SIZE + ty;
  int Col = get_group_id(0) * BLOCK_SIZE + tx;

  float value = 0.0;
  int m = 0;

  __local float aLocal[BLOCK_SIZE][BLOCK_SIZE];
  __local float bLocal[BLOCK_SIZE][BLOCK_SIZE];

  for ( int i = 0; i < mSize ; i += BLOCK_SIZE ) {
    if ( Row >= mSize && Col >= mSize ) {
      aLocal[ty][tx] = 0.0f;
      bLocal[ty][tx] = 0.0f;
    }
    else {
      aLocal[ty][tx] = A[ mSize * Row + (i + tx)];
      bLocal[ty][tx] = B[ mSize * (i + ty) + Col];
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    if ( Row < mSize && Col < mSize ) {
      m = ( (mSize - i) < BLOCK_SIZE ) ?
        (mSize - i): BLOCK_SIZE;

      for (int j = 0; j < m; j++)
        value += aLocal[ty][j] * bLocal[j][tx];
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    if ( Row < mSize && Col < mSize )
      C[Row * mSize + Col] = value;
    }
}
```