An Introduction to the OpenCL Programming Model

Jonathan Tompson*
NYU: Media Research Lab

Kristofer Schlachter†
NYU: Media Research Lab

Abstract

This paper presents an overview of the OpenCL 1.1 standard [Khronos 2012]. We first motivate the need for GPGPU computing and then discuss the various concepts and technological background necessary to understand the programming model. We use concurrent matrix multiplication as a framework for explaining various performance characteristics of compiling and running OpenCL code, and contrast this to native code on more traditional general purpose CPUs.

Keywords: OpenCL, Matrix Multiply, Barrier Synchronization

1 Introduction

In recent years performance scaling for general purpose CPUs has failed to increase as predicted by Gordon Moore in the early 1970s [Sutter 2005], and therefore raw throughput for sequential code has plateaued between subsequent processor generations. As a result of the issues of working with deep sub-micron lithography[1], the primary motivation for moving to new processing nodes is less about increased performance or efficiency, but rather economic costs (for instance decreased cost per transistor and larger wafer sizes). While we have seen modest performance increases from the latest microprocessor architectures from Intel and AMD, these certainly haven't resulted in the doubling of performance that computer scientists relied upon from the 70's through to the early 2000's, in order to see performance increases in their code without changing the top level execution model.

Motivated by this lack of performance scaling, Graphics Processing Unit (GPU) manufacturers have opened up low level hardware traditionally used for graphics rendering only, in order to perform highly parallel computation tasks on what they call General Purpose GPU cores (GPGPU). While low level GPGPU execution cores lack branch prediction and out of order execution hardware that allow traditional superscalar CPU architectures to optimize sequential code, moving computation to the GPU trades off flexibility in execution models for raw performance. More-recently, CUDA[2] and OpenCL[3] are two frameworks that have seen significant traction and adoption by third-party software developers. This paper will focus on OpenCL (specifically version 1.1 of the specification) since it is an open cross-platform & cross-vendor standard. The paper is not a thorough investigation into the OpenCL standard (which is itself a massive body of work), but is an overview of the programming methodologies one should be aware of when considering writing GPGPU code.

*e-mail:tompson@cims.nyu.edu
†e-mail:ks228@cs.nyu.edu

[1]From the author's experience the most notable of these issues include: worsened short channel effects, an increase in the ratio of extrinsic parasitics vs transistor transconductance, limits on unity gain frequency scaling, and sub-threshold and gate oxide leakage

[2]from NVIDIA - first released 2006

[3]originally developed by Apple but now managed by the non-profit technology consortium Khronos Group - first released 2008

2 The OpenCL Standard

2.1 OpenCL Architecture and Hardware

OpenCL is a programming framework and standard set from Khronos, for heterogeneous parallel computing on cross-vendor and cross-platform hardware. It provides a top level abstraction for low level hardware routines as well as consistent memory and execution models for dealing with massively-parallel code execution. The advantage of this abstraction layer is the ability to scale code from simple embedded microcontrolers to general purpose CPUs from Intel and AMD, up to massively-parallel GPGPU hardware pipelines, all without reworking code. While the OpenCL standard allows OpenCL code to execute on CPU devices, this paper will focus specifically on using OpenCL with Nvidia and ATI graphics cards as this represents (in the authors opinion) the pinnacle of consumer-level high-performance computing in terms of raw FLOPS throughput, and has significant potential for accelerating "suitable" parallel algorithms.

Figure 1 shows an overview of the OpenCL architecture. One CPU-based "Host" controls multiple "Compute Devices" (for instance CPUs & GPUs are different compute devices). Each of these coarse grained compute devices consists of multiple "Compute Units" (akin to execution units & arithmetic processing unit groups on multi-core CPUs - think "cores") and within these are multiple "Processing Elements". At the lowest level, these processing elements all execute OpenCL "Kernels" (more on this later).

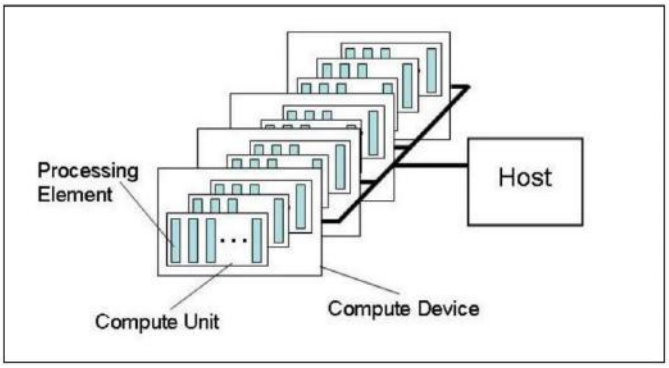

Figure 1: *OpenCL Platform Model (from [Khronos 2011])*

The specific definition of compute units is different depending on the hardware vendor. In AMD hardware, each compute unit contains numerous "stream cores" (or sometimes called SIMD Engines) which then contain individual processing elements. The stream cores are each executing VLIW 4 or 5 wide SIMD instructions. See figure 2 for an overview of ATI hardware. In NVIDIA hardware they call compute units "stream multiprocessors" (SM's) (and in some of their documentation they are refereed to as "CUDA cores"). In either case, the take away is that there is a fairly complex hardware hierarchy capable of executing at the lowest level SIMD VLIW instructions.

An important caveat to keep in mind is that the marketing numbers for core count for NVIDIA and ATI aren't always a good rep-

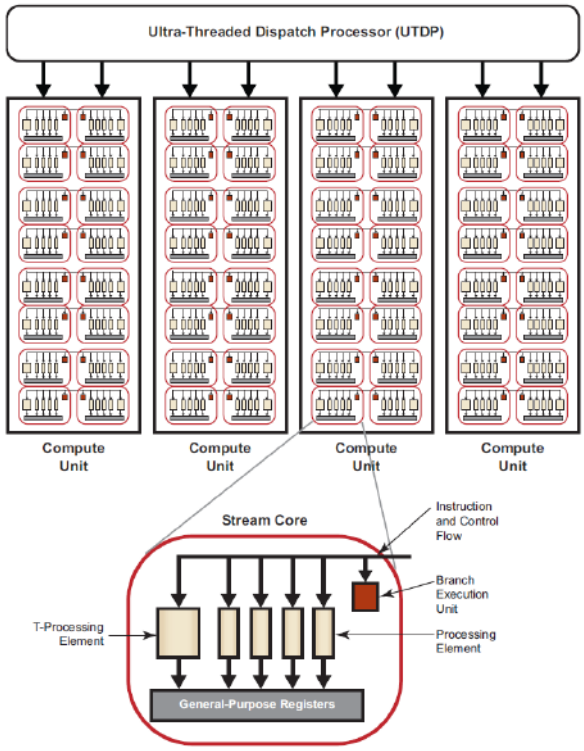

Figure 2: *Simplified block diagram of ATI compute device (from [ATI 2011])*

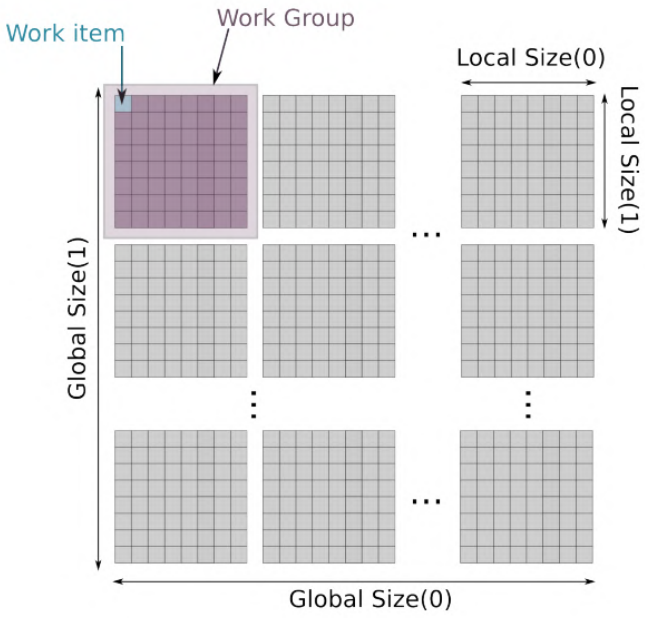

Figure 3: *2D Data-Parallel execution in OpenCL (from [Boydstun 2011])*

resentation of the capabilities of the hardware. For instance, on NVIDIA's website a Quadro 2000 graphics card has 192 "Cuda Cores". However, we can query the lower-level hardware capabilities using the OpenCL API and what we find is that in reality there are actually 4 compute units, all consisting of 12 stream multiprocessors, and each stream multiprocessor is capable of 4-wide SIMD. $192 = 4*12*4$. In the author's opinion this makes the marketing material confusing, since you wouldn't normally think of a hardware unit capable only of executing floating point operations as a "core". Similarly, the marketing documentation for a HD6970 (very high end GPU from ATI at time of writing) shows 1536 processing elements, while in reality the hardware has 24 compute units (SIMD engines), and 16 groups of 4-wide processing elements per compute unit. $1536 = 24 * 16 * 4$.

2.2 OpenCL Execution Models

At the top level the OpenCL host [4] uses the OpenCL API platform layer to query and select compute devices, submit work to these devices and manage the workload across compute contexts and work-queues. In contrast, at the lower end of the execution hierarchy (and at the heart of all OpenCL code) are OpenCL "Kernels" running on the each processing element. These Kernels are written in OpenCL C [5] that execute in parallel over a predefined N-dimensional computation domain. In OpenCL vernacular, each independent element of execution in this domain is called a "work-item" (which NVIDIA refers to as "CUDA threads"). These work-items are grouped together into independent "work-groups" (which NVIDIA refers to as a "thread block"). See Figure 3 for a top level overview of this structure.

[4] in our case written in C++, though other language bindings exist

[5] OpenCL C is a subset of C99 with appropriate language additions

According to the documentation, the execution model is "fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism" [NVIDIA 2012]. Data-parallel programming is where the domain of execution for each thread is defined by some region over a data structure or memory object (typically a range of indices into an N-by-N array as depicted by Figure 3), where execution over these sub-regions are deemed independent. The alternative model is task-parallel programming, whereby concurrency is exploited across domains of task level parallelism. OpenCL API exploits both of these, however since access to global memory is slow one must be careful in writing Kernel code that reflects the memory access performances of certain memory locations in the hierarchy (more on memory hierarchy later). In this way work-groups can be separated by task-parallel programming (since threads within a work-group can share local memory), but are more likely sub-domains in some larger data structure as this benefits hardware memory access (since getting data from DRAM to global GPU memory is slow, as is getting data from global GPU memory to local work-group memory).

Since hundreds of threads are executed concurrently which results in a linear scaling in instruction IO bandwidth, NVIDIA uses a SIMT (Single-Instruction, Multiple-Thread) architecture. One instruction call in this architecture executes identical code in parallel by different threads and each thread executes the code with different data. Such a scheme reduces IO bandwidth and allows for more compact thread execution logic. ATI's architecture follows a very similar model (although the nomenclature is different).

With the framework described above, we can now outline the basic pipeline for a GPGPU OpenCL application.

1. Firstly, a CPU host defines an N-dimensional computation domain over some region of DRAM memory. Every index of this N-dimensional computation domain will be a work-item and each work-item executes the same Kernel.
2. The host then defines a grouping of these work-items into work-groups. Each work-item in the work-groups will exe-

cute concurrently within a compute unit (NVIDIA streaming multiprocessor or ATI SIMD engines) and will share some local memory (more later). These work-groups are placed onto a work-queue.

3. The hardware will then load DRAM memory into the global GPU RAM and execute each work-group on the work-queue.
4. On NVIDIA hardware the multiprocessor will execute 32 threads at once (which they call a "warp group"), if the work-group contains more threads than this they will be serialized, which has obvious implications on the consistency of local memory.

Each processing element executes purely sequential code. There is no branch prediction and no speculative execution, so that all instructions in a thread are executed in order. Furthermore, some conditional branch code will actually require execution of both branch paths, which are then data-multiplexed to produce a final result. I will refer the reader to the Khronos OpenCL, ATI and NVIDIA documentations for further details since the details are often complicated. For instance, a "warp" in NVIDIA hardware executes only one common instruction at a time on all threads in the work-group (since access to individual threads is through global SIMT instructions), so full efficiency is only realized when all 32 threads in the warp agree on their execution path.

There are some important limitations on work-groups to always keep in mind. Firstly, the global work size must be a multiple of the work-group size, or another way of saying that is that the work-groups must fit evenly into the entire data structure. Secondly, the work-group size (which of a 2D array would be the $size^2$) must be less than or equal to the CL_KERNEL_WORK_GROUP_SIZE flag. This is a hardware flag stating the limitation on the maximum concurrent threads within a work-group. OpenCL will return an error code if either of these conditions are violated [6].

2.3 OpenCL Memory Model

The OpenCL memory hierarchy (shown in Figure 4) is structured in order to "loosely" resemble the physical memory configurations in ATI and NVIDIA hardware. The mapping is not 1 to 1 since NVIDIA and ATI define their memory hierarchies differently. However the basic structure of top global memory vs local memory per work-group is consistent across both platforms. Furthermore, the lowest level execution unit has a small private memory space for program registers.

These work-groups can communicate through shared memory and synchronization primitives, however their memory access is independent of other work-groups (as depicted in Figure 5). This is essentially a data-parallel execution model, where the domain of independent execution units is closely tied and defined by the underlining memory access patterns. For these groups, OpenCL implements a relaxed consistency, shared memory model. There are exceptions, and some compute devices (notably CPUs) can execute task-parallel compute Kernels, however the bulk of OpenCL applications on GPGPU hardware will execute strictly data-parallel workers.

An important issue to keep in mind when programming OpenCL Kernels is that memory access on the DRAM global and local memory blocks is not protected in any way. This means that segfaults are not reported when work-items dereference memory outside their own global storage. As a result, GPU memory set aside for the OS can be clobbered unintentionally, which can result in behaviors

[6] In general, if you don't check the return conditions for all the API functions then the Kernel will either cause the host program to crash or crash your OS. Always check error flags!

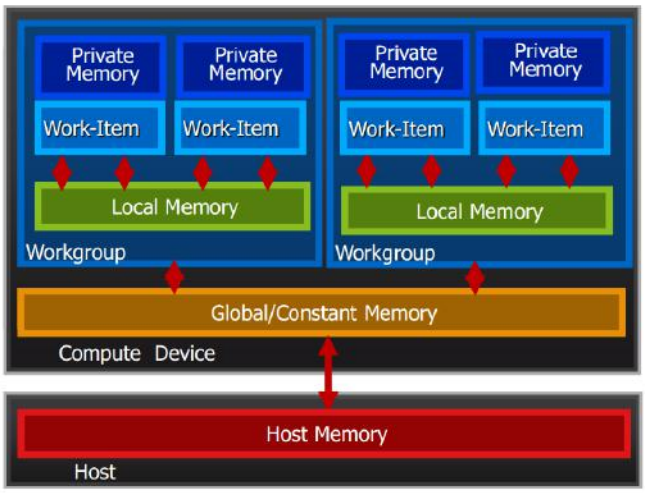

Figure 4: *OpenCL Memory Model (from [Khronos 2011])*

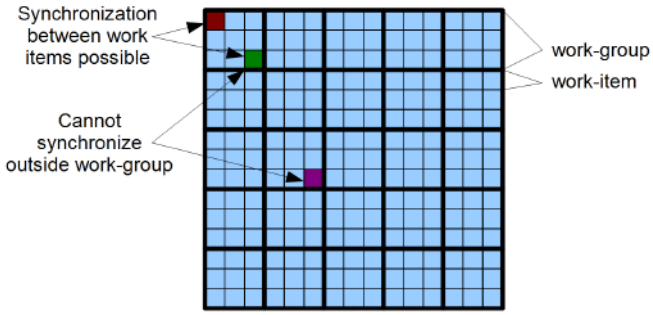

Figure 5: *OpenCL Work-group / Work-unit structure*

ranging from benign screen flickering up to frustrating blue screens of death and OS level crashes.

Another important issue is that mode-switches may result in GPU memory allocated to OpenCL to be cannibalized by the operating system. Typically the OS allocates some portion of the GPU memory to the "primary-surface", which is a frame buffer store for the rendering of the OS. If the resolution is changed during OpenCL execution, and the size of this primary-surface needs to grow, it will use OpenCL memory space to do so. Luckily these events are caught at the driver level and will cause any call to the OpenCL runtime to fail and return an invalid context error.

Memory fences are possible within threads in a work-group as well as synchronization barriers for threads at the work-item level (between individual threads in a processing element) as well as at the work-group level (for coarse synchronization between work-groups). On the host side, blocking API functions can perform waits for certain events to complete, such as all events in the queue to finish, specific events to finish, etc. Using this coarse event control the host can decide to run work in parallel across different devices or sequentially, depending on how markers are placed in the work-queue (as depicted in Figure 6).

Finally, you should also be careful when statically allocating local data (per work-group). You should check the return conditions from the host API for flags indicating that you're allocating too much per work-group, however you should also be aware that sometimes the

Kernel will compile anyway and will result in a program crash [7].

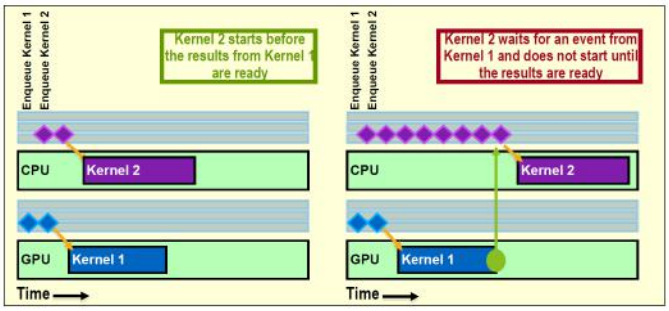

Figure 6: *Concurrency control with OpenCL event-queueing*

3 Matrix Multiply Example

3.1 CPU Implementation

Matrix multiplication is an obvious example for data-parallel concurrency optimizations, since input data is unmodified and the output data range consists of a set of independent computation tasks. Naive matrix multiply algorithms are $O\left(n^3\right)$, and consist of a simple triple for-loop; the two outer loops iterate over the row and column index and the inner for loop performs a dot-product of the row & column vectors. Optimizations for sequential code on a CPU include cache line pre-fetching and other "cache aware data access" techniques, as well as the use of SSE SIMD instructions for modest speed gains. *cpu.cpp* is a very simple implementation for matrix multiply on the CPU:

```
// CPU matrix multiply C = A * B
void matMul(float* A, float* B, float* C, int dim) {
  for (int row = 0; row < dim; row ++) {
    for (int col = 0; col < dim; col ++) {
      // Dot row from A with col from B
      float val = 0;
      for (int i = 0; i < dim; i ++)
        val += A[row * dim + i] * B[i * dim + col];
      C[row*dim + col] = val;
    }
  }
}
```

cpu.cpp

The above code was compiled and run on three different machines[8]; one laptop running Mac OS X and compiling with gcc, and two desktops running Windows 7 and compiling with Microsoft Visual Studio compiler. Figure 7 shows the performance vs matrix dimension. The performance is not linear in the loglog plot (therefore strictly not polynomial time) which has to do with cache line thrashing. Since the two vectors are stored row major at least one of the matrices has to be read sequentially across the columns (which may not exist in the same cache block) when performing the dot products. There are many techniques for improving cache performance, however, since the focus of this paper is not optimizing single threaded matrix multiply, we leave this up to the reader to investigate the standard references, and we present this data as a frame of reference only for comparison with OpenCL results.

[7] In general we found that on Mac OS X Lion using ATI hardware these sort of crashes were more likely.

[8] Please see the Appendix for details

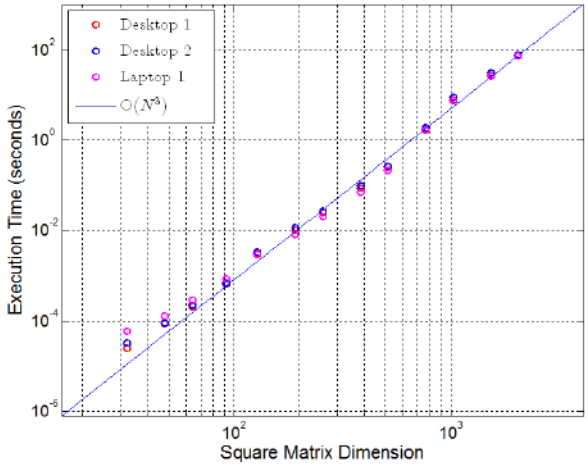

Figure 7: *cpu.cpp Performance vs matrix dimensions*

Better algorithms include Strassen's algorithm [Huss-Lederman et al. 1996][9] which is $O\left(n^{log_2 7}\right) \approx O\left(n^{2.807}\right)$. The best known polynomial time algorithm is the Coppersmith-Winograd algorithm [Coppersmith and Winograd 1987] and has asymptotic complexity of $O\left(n^{2.373}\right)$. However, the constant factors for these divide-and-conquer approaches are prohibitively high for even reasonably sized matrices.

As an extension of the naive implementation above, we can take advantage of the fact that modern cpu's have multiple cores. In cpu_mt.cpp below, each thread only calculates and writes a subset of the output matrix. This means that threads do not need to synchronize their writes. Since the read access patters are the same as the naive implementation it will still suffer from the same cache performance issues. Figure 8 shows the measured results of cpu_mt.cpp. For large matrix dimensions the multi-threaded approach achieves roughly 4x speedup on a 4 core machine, however for small matrices the overhead of spawning threads dominates the runtime performance.

```
void *threadFunc(void* arg) {
  for (int row = startRow; row < endRow; row ++) {
    for (int col = 0; col < dim; col ++) {
      // Dot row from A with col from B
      float val = 0;
      for (int i = 0; i < dim; i ++) {
        val += A[row * dim + i] * B[i * dim + col];
      }
      C[row*dim + col] = val;
    }
  }
}
```

cpu_mt.cpp

3.2 Naive OpenCL Kernel

The obvious alternative, is to let our massively parallel OpenCL architecture execute simple concurrent Kernels, each of which performs one dot-product associated with one output matrix element. Doing so does not change the asymptotic complexity, however we can reduce the constant factor significantly. Our first attempt at a

[9] Although this algorithm suffers from numerical stability issues it is a popular approach and is available in many linear algebra libraries (notably BLAS).

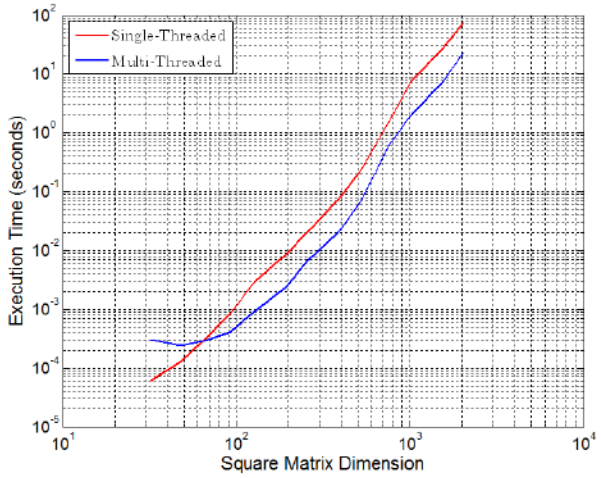

Figure 8: *cpu.cpp Single-Threaded vs Multi-Threaded performance*

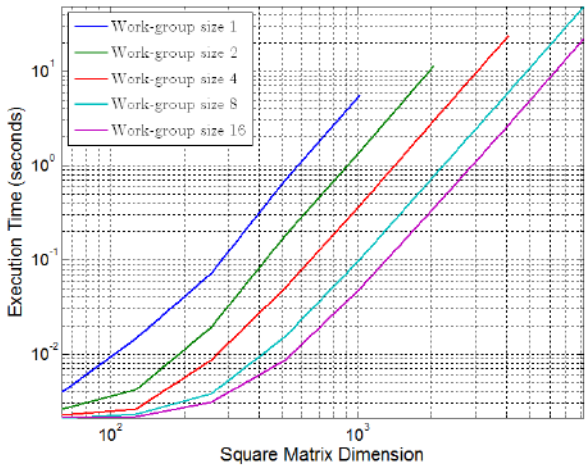

Figure 9: *matmul1.cl Performance vs work-group size and matrix dimensions*

matrix multiply OpenCL Kernel is depicted in *matmul1.cl*. This approach is simple, each thread reads the x row of matrix A and the y column of matrix B from the global GPU memory and computes the corresponding (x, y) element of the output matrix C.

```
// OpenCL Kernel for matrix multiply. C = A * B
__kernel void
matrixMul(__global float* C,
          __global float* A,
          __global float* B,
          int wA, int wB) {
  int tx = get_global_id(0);  // 2D Thread ID x
  int ty = get_global_id(1);  // 2D Thread ID y

  // Perform dot-product accumulate into value
  float value = 0;
  for (int k = 0; k < wA; ++k) {
    value += A[ty * wA + k] * B[k * wB + tx];
  }
  C[ty * wA + tx] = value;  // Write to device memory
}
```

matmul1.cl

Figure 9 shows the performance of *matmul1.cl* vs the matrix dimension for different values of the tunable parameter: work-group size. As explained in Section 2, each thread in a work-group executes in parallel, so we expect larger work-group sizes to take advantage of maximum concurrency. If the work-group size is larger than the vector width of a compute unit core, the Work-Group is executed in sequential groups of the maximum width. The surprising result here is that even work-group sizes that are much larger than the maximum concurrent thread size[10], serialization of the work-group execution doesn't affect performance and actually improves it. This is because the overhead of switching groups of work-items (NVIDIA calls these warps, while ATI calls these wavefronts) of serial execution is much faster than switching the memory registers for each work-group (since the GPU stores all work-group registers before retiring them). The very important take-away is that you should always try and keep as many threads per work-group as possible to maximize performance.

[10]The maximum number of concurrent threads on our NVIDIA hardware is 32. Since the matrix array is 2D, the number of threads is the work-group size squared. So in Figure 9, work-group size = 16 results in 256 threads

To make sure that these results were consistent across different GPUs, the code was compiled and run on our three test systems. The results are shown in Figure 10. The CPU performance results are also plotted for reference. What is clear from these results is that the overhead of moving data on and off the GPU (including the matrices themselves, the Kernel instructions, and the work-queue information) is significant. As such, for small matrix sizes the CPU actually outperforms the GPU as we might expect. Another issue worth noting is that the GPU overhead is platform dependant. Informally, we have found that running the OpenCL code on ATI cards the overhead is higher than on NVIDIA hardware.

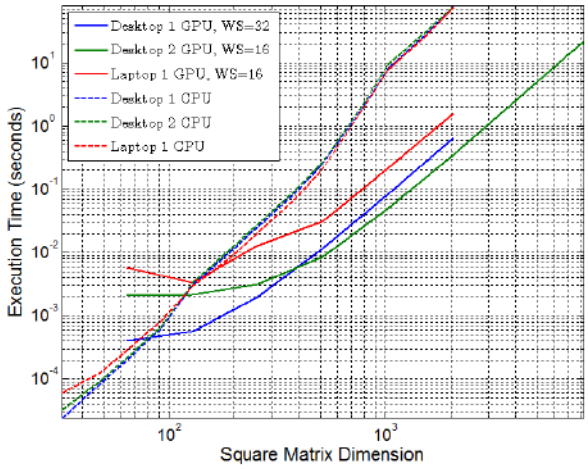

Figure 10: *matmul1.cpp performance over hardware types*

3.3 Shared Memory OpenCL Kernel

One way to increase the speed of execution for each thread is to take advantage of the faster memory types in the OpenCL memory hierarchy. As already mentioned, the slowest memory is the global memory available to all threads in all work-groups. The next fastest memory type is local memory that is only visible to threads of the work-group it belongs to. A loose analogy is that the global

memory to local memory relationship is akin to CPU RAM and private L1 cache. *matmul2.cl* is an OpenCL Kernel that tries to take advantage of the varying speeds in this memory hierarchy.

Unlike global memory, local memory isn't initialized and transferred to the device from the host. The work-item threads must fill local memory before they can use it and therefore this is the first computational task in *matmul2.cl*. The Kernel fills one location per thread and then a synchronization barrier waits for all of the other threads to fill up the rest of the data. Since the local store has limited size, *matmul2.cl* implements block matrix multiplication to address smaller subsets of the global domain, but needs to iterate over multiple sub-blocks in order to produce a correct matrix multiply. The memory write pattern is therefore heavy on local memory and only writes once to the global memory.

```
// OpenCL Kernel for BLOCK matrix multiply. C = A * B
__kernel void
matrixMul(__global float* C,
          __global float* A,
          __global float* B,
          int wA, int wB) {
  int bx = get_group_id(0);   // 2D Thread ID x
  int by = get_group_id(1);   // 2D Thread ID x
  int tx = get_local_id(0);   // 2D local ID x
  int ty = get_local_id(1);   // 2D local ID y

  // first and last sub-matrix of A for this block
  int aBegin = wA * BLOCK_SIZE * by;
  int aEnd   = aBegin + wA - 1;
  int aStep  = BLOCK_SIZE;

  // first and last sub-matrix of B for this block
  int bBegin = BLOCK_SIZE * bx;
  int bStep  = BLOCK_SIZE * wB;

  float Csub = 0.0;
  // Iterate over all sub-matrices of A and B
  for (int a = aBegin, b = bBegin; a <= aEnd; a+=aStep,
                                              b+=bStep) {
    // Static work-group local allocations
    __local float As[BLOCK_SIZE][BLOCK_SIZE];
    __local float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Each thread loads one element of the block
    // from global memory
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    // Barrier to synchronize all threads
    barrier(CLK_LOCAL_MEM_FENCE);
    // Now the local sub-matricies As and Bs are valid

    // Multiply the two sub-matrices. Each thread
    // computes one element of the block sub-matrix.
    for (int k = 0; k < BLOCK_SIZE; ++k)
      Csub += As[ty][k] * Bs[k][tx];

    // Barrier to synchronize all threads before moving
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;  // write to global memory
}
```

matmul2.cl

Figure 11 shows the incredible performance improvement associated with moving data to local storage. The reader should note that for smaller work-group sizes the performance improvement is less. This is because the overhead of context-switching local memory blocks between work-group executions is high.

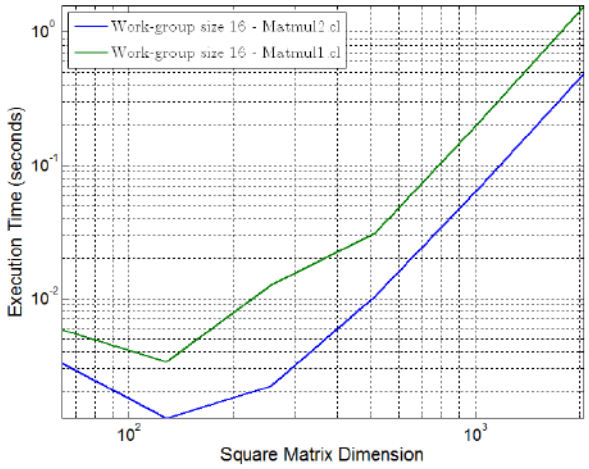

Figure 11: matmul1.cl *vs* matmul2.cl

4 Conclusion

Figure 12 shows the comparative performance of the best CPU results and the best GPU OpenCL results. Clearly, for small matrices the overhead of OpenCL execution dominates the performance benefits of massivly concurrent execution. For our measurements, below a matrix dimension of roughly 150 x 150 the simple multi-threaded CPU code out performs OpenCL.

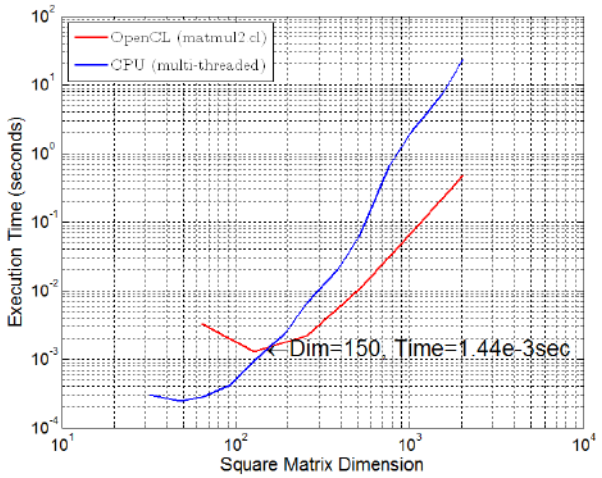

Figure 12: *Best OpenCL Vs. CPU performance*

At this point the authors would like to reiterate the insights we have gained from our short experience with OpenCL. Firstly, since context switch overheads between warps within work-groups is low, care must be taken to make sure that all compute-units are running at full capacity, and that the work-group size is as large as possible. Said in another way, large work-groups reduce the effects of memory latency. Secondly, memory locality is extremely important. This is intuitive since the number of execution threads is extremely large (in comparison to conventional CPU multi-threaded environments), and therefore the performance impact of sharing external global memory (GDDR5 in our hardware) is large due to high memory contention, and also since local memory resides in

extremely fast on-chip SRAM in most cases.

Another issues that is not well represented by the content of this paper alone, is that there is a huge code overhead on the Host side with setting up and running OpenCL code. Even though our smallest Kernel size might be only 15 lines, there is an additional 1500 lines of code on the Host side in support. In our opinion this is one of the biggest drawbacks to the OpenCL platform, as is familiarizing oneself with the considerable complexity of OpenCL API and execution models in general.

As a side note, as a result of our experiences with the OpenCL platform, as well as having spoken with colleagues expertly familiar with GPGPU programming, our belief is that the OpenCL toolkits are less mature. We experienced many unexplained driver crashes (though sometimes the fault of our own), and the Khronos documentation isn't as clear as we feel it should be. Hopefully, the platform will improve in the near future.

Acknowledgements

We would like to thank Eric Hielsher for his invaluable insights into GPU architectures. Also Starbucks was very helpful ⌣.

References

ATI, 2011. Programming guide: Amd accelerated parallel processing. http://developer.amd.com/sdks/amdappsdk/assets/amd_accelerated_parallel_processing_opencl_programming_guide.pdf.

BOYDSTUN, K., 2011. Introduction opencl (caltech lecture). http://www.tapir.caltech.edu/~kboyds/OpenCL/opencl.pdf.

COPPERSMITH, D., AND WINOGRAD, S. 1987. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, ACM, New York, NY, USA, STOC '87, 1–6.

HUSS-LEDERMAN, S., JACOBSON, E. M., TSAO, A., TURNBULL, T., AND JOHNSON, J. R. 1996. Implementation of strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, IEEE Computer Society, Washington, DC, USA, Supercomputing '96.

KHRONOS, 2011. Opencl overview. http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf/.

KHRONOS, 2012. Opencl 1.2 reference pages. http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/.

NVIDIA, 2012. Opencl programming guide for the cuda architecture, version 4.2. http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf.

SUTTER, H. 2005. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*.

Appendix

Computers Used When Profiling

Device	CPU	GPU	RAM	GRAM	Max WG size	Compute Units	Proc. Elem.	Local mem size
Desktop 1	Intel Xeon W3550	NVIDIA Quadro 2000	18GB	1024MB	1024	4	192	49152
Desktop 2	Intel Core i7 930	ATI Radeon HD 5850	6GB	1024MB	256	18	1440	32768
Laptop 1	Intel Core i7 2820QM	ATI Radeon HD 6750M	4GB	1024MB	1024	6	720	32768