From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming[☆,☆☆]

Peng Du[a], Rick Weber[a], Piotr Luszczek[a], Stanimire Tomov[a], Gregory Peterson[a], Jack Dongarra[a,b]

[a] *University of Tennessee Knoxville*
[b] *University of Manchester*

Abstract

In this work, we evaluate OpenCL as a programming tool for developing performance-portable applications for GPGPU. While the Khronos group developed OpenCL with programming portability in mind, performance is not necessarily portable. OpenCL has required performance-impacting initializations that do not exist in other languages such as CUDA. Understanding these implications allows us to provide a single library with decent performance on a variety of platforms. We choose triangular solver (TRSM) and matrix multiplication (GEMM) as representative level 3 BLAS routines to implement in OpenCL. We profile TRSM to get the time distribution of the OpenCL runtime system. We then provide tuned GEMM kernels for both the NVIDIA Tesla C2050 and ATI Radeon 5870, the latest GPUs offered by both companies. We explore the benefits of using the texture cache, the performance ramifications of copying data into images, discrepancies in the OpenCL and CUDA compilers' optimizations, and other issues that affect the performance. Experimental results show that nearly 50% of peak performance can be obtained in GEMM on both GPUs in OpenCL. We also show that the performance of these kernels is not highly portable. Finally, we propose using auto-tuning to better explore these kernels' parameter space using search heuristics.

Keywords: hardware accelerators, portability, auto-tuning

1. Introduction

People associated Graphics Processing Units (GPUs) with fast image rendering until the turn of the century. This is when the science community turned their attention to the hardware predominantly discussed in the computer gaming circles. One of the first attempts of non-graphical computations on a GPU was a matrix-matrix multiply [1]. In 2001, low-end graphics cards had no floating-point support; floating-point color buffers did not arrive until 2003 [2]. For the gaming industry, support for floating-point meant more realistic game-play; rich advanced lighting effects no longer suffered from banding effects common in older generations of hardware that only allowed a single byte per color channel. For the scientific community, the addition of floating point meant that overflow associated with fixed-point arithmetic was no longer a problem. Many research publications thoroughly analyzed the new breed of GPU hardware using languages borrowed from the graphics community [3, 4, 5]. It goes without saying that these computational advances would not be possible if it weren't for the programmable shaders that broke the rigidity of the fixed graphics pipeline. LU factorization with partial pivoting on a GPU was one of the first common computational kernels that ran faster than an optimized CPU implementation [6]. The introduction of NVIDIA's CUDA [7, 8] (Compute Unified Device Architecture), ushered a new era of improved performance for many applications as programming GPUs became simpler: archaic terms such as texels, fragments, and pixels were superseded with threads, vector processing, data caches and shared memory.

Further changes occuring in ATI's and NVIDIA's offerings made GPU acceleration even more pertinent to the scientific community. ATI's FireStream and NVIDIA's Fermi architecture added support for Fused Multiply-Add (FMA): a more accurate version of the former MAD (Multiply-Add) instruction. With only a single rounding step, this new instructions bring GPUs even closer to compliance with the IEEE 754 standard for floating-point arithmetic. Additionally, reworking of the cache hierarchy helped with some of the performance issues of the past. Finally, Error Correction Codes (ECC) are now used to protect the GPU device's memory as its capacity grows to the point of being vulnerable to errors induced by nature, such as cosmic radiation.

In our project called Matrix Algebra on GPU and Multicore Architectures [9] (MAGMA), we mainly focus on dense matrix routines for numerical linear algebra, similar to those available in LAPACK [10]. While CUDA is only available for NVIDIA GPUs, there are other existing frameworks that allow platform-independent programming for GPUs:

1. DirectCompute from Microsoft,
2. OpenGL Shading Language (GLSL), and
3. OpenCL.

[☆] This work was supported by the SCALE-IT fellowship through grant number OR11907-001.

[☆☆] This work was supported by NVIDIA, Microsoft, the U.S. National Science Foundation, and the U.S. Department of Energy.

DirectCompute allows access to graphics cards from multiple vendors. However, it is specific to Microsoft Windows and therefore it is not portable between host Operating Systems (OS).

The OpenGL Shading language [11] is portable across both GPU hardware and the host OS. However, it is specifically geared towards programming new graphics effects – GLSL does not have a scientific focus.

OpenCL [12] has been designed for general purpose computing on GPUs (GPGPU). It is an open standard maintained by the Khronos group with the backing of major graphics hardware vendors as well as large computer industry vendors interested in off-loading computations to GPUs. As such, there exist working OpenCL implementations for graphics cards and, multi-core processors; OpenCL offers portability across GPU hardware, OS software, and multicore processors. In this work, BLAS routines are used to evaluate OpenCL's usefulness in creating a portable high performance GPU numerical linear algebra library.

The rest of the paper is organized as follows: Section 2 talks about related work. Section 3 details OpenCL's programming considerations. This involves a comparison between CUDA and OpenCL as programming language for GPU, and the profiling analysis of the run-time of each component in OpenCL program. Sections 4 and 5 give an evaluation to both NVIDIA and ATI GPU platforms by implementing fast GEMM and analyzing performance. Section 6 discusses cross-platform issues and section 7 lays out the basic structure of an auto-tuning system for cross-platform GPU math library design. Section 8 shows the performance result and section 9 concludes the paper. In the text we use the terms "ATI card" and Radeon 5870; "Fermi card" and Tesla C2050 interchangeably.

2. Related Work

Synthetic benchmarking of GPUs was used extensively to gain understanding of the graphics accelerators when the technical details of the hardware remains an industrial secret [13]. In the context of scientific applications, such benchmarking efforts lead to algorithms that provide significant performance improvements [14].

A performance-oriented study of NVIDIA's PTX (Parallel Thread eXecution) architecture [15] was done in the context of the Ocelot project [16]. Code transformations of CUDA kernels were done by the MCUDA [17] project. The transformations enabled CUDA code to run efficiently on multicore CPUs. The inverse operation – porting multicore code to GPUs – was difficult [16].

Work on optimizing CUDA implementations of basic linear algebra kernels has demonstrated the importance on "how sensitive the performance of a GPU is to the formulation of your kernel" [18] and that an enormous amount of well thought experimentation and benchmarking [14, 18] is needed in order to optimize performance. Optimizing OpenCL applications for a particular architecture faces the same challenges. Further, optimizing a fixed OpenCL code for several architectures is harder, even impossible, and naturally, many authors claim that OpenCL does not provide performance portability. This, along with the fact that GPUs are quickly evolving in complexity, has made tuning numerical libraries for them challenging. One approach (that we explore) to systematically resolve these issues is the use of auto-tuning, a technique that in the context of OpenCL would involve collecting and generating multiple kernel versions, implementing the same algorithm optimized for different architectures, and heuristically selecting the best performing one. Auto-tuning has been used intensively on CPUs in the past to address these challenges to automatically generate near optimal numerical libraries, e.g., ATLAS [19, 20] and PHiPAC [21] used it to generate highly optimized BLAS. Work on auto-tuning CUDA kernels for NVIDIA GPUs [22, 23] has shown that the technique is a very practical solution to easily port existing algorithmic solutions on quickly evolving GPU architectures and to substantially speed up even highly tuned hand-written kernels.

In [24], the authors[1] examined performance portability in OpenCL. In their study, they compared CUDA and OpenCL implementations of a Monte Carlo Chemistry application running on an NVIDIA GTX285. They also compared the same application written in ATI's now defunct Brook+ to an OpenCL version on a Firestream 9170 and Radeon 4870 respectively. Finally they compared OpenCL to a C++ implementation running on multi-core Intel processors. The paper showed that while OpenCL does provide code portability, it doesn't necessarily provide performance portability. Furthermore, they showed that platform-specific languages often, but not always, outperformed OpenCL.

3. OpenCL as A Programming Tool

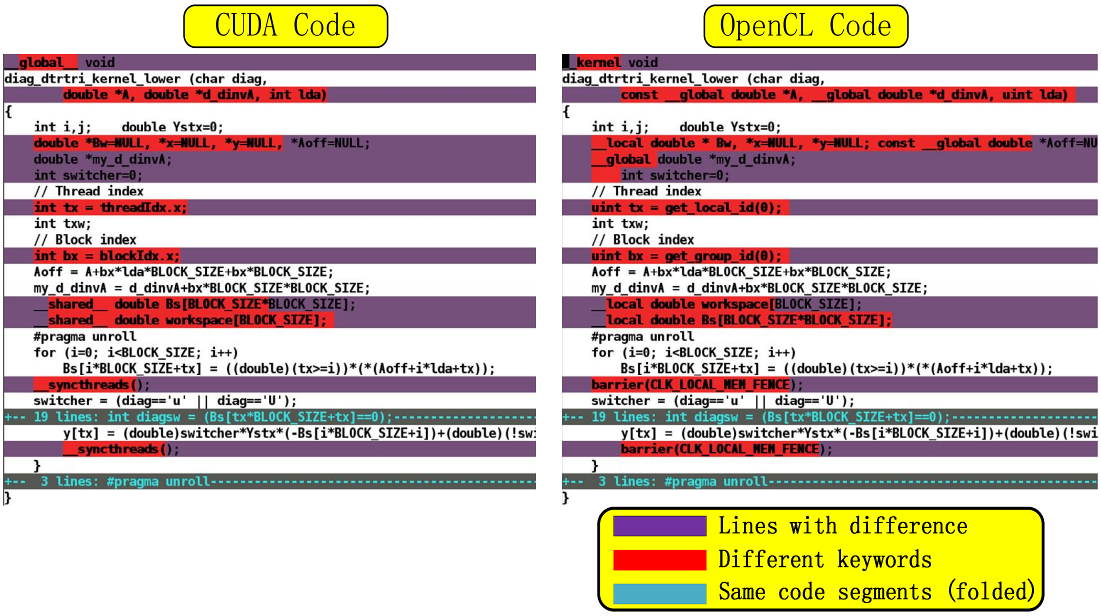
To evaluate OpenCL as a programming tool for implementing high performance linear algebra routines, we pick the triangular solver (TRSM) routine from BLAS and profile each component of the program: environment setup, program compilation, kernel extraction, and execution.

TRSM solves the linear equation $Ax = b$ where A is an upper or lower triangular matrix and b is a known matrix of solutions. Its implementation involves a blocking algorithm in which the diagonal triangular blocks are inverted (TRTRI) in parallel followed by a series of matrix multiplications (GEMM). Porting these routines from CUDA to OpenCL requires some translation.

CUDA and OpenCL have many conceptual similarities but they diverge on terminology. Table 1 shows the corresponding terms in both frameworks while 1 highlights differences in the CUDA and OpenCL software stacks. Similarly, ATI and NVIDIA GPUs have analogous platform definitions as shown in Table 2. Table 3 shows the platform details of two different NVIDIA GPUs and one GPU from ATI/AMD. We show what these differences mean to application developers.

---

[1] Rick Weber and Gregory Peterson are also authors of this paper



CUDA Code

```
__global__ void
diag_dtrtri_kernel_lower (char diag,
        double *A, double *d_dinvA, int lda)
{
    int i,j;    double Ystx=0;
    double *Bw=NULL, *x=NULL, *y=NULL, *Aoff=NULL;
    double *my_d_dinvA;
    int switcher=0;
    // Thread index
    int tx = threadIdx.x;
    int txw;
    // Block index
    int bx = blockIdx.x;
    Aoff = A+bx*lda*BLOCK_SIZE+bx*BLOCK_SIZE;
    my_d_dinvA = d_dinvA+bx*BLOCK_SIZE*BLOCK_SIZE;
    __shared__ double Bs[BLOCK_SIZE*BLOCK_SIZE];
    __shared__ double workspace[BLOCK_SIZE];
    #pragma unroll
    for (i=0; i<BLOCK_SIZE; i++)
        Bs[i*BLOCK_SIZE+tx] = ((double)(tx>=i))*(*(Aoff+i*lda+tx));
    __syncthreads();
    switcher = (diag=='u' || diag=='U');
+-- 19 lines: int diagsw = (Bs[tx*BLOCK_SIZE+tx]==0);-------------------
        y[tx] = (double)switcher*Ystx*(-Bs[i*BLOCK_SIZE+i])+(double)(!sw
        __syncthreads();
    }
+--  3 lines: #pragma unroll--------------------------------------------
}
```

OpenCL Code

```
█_kernel void
diag_dtrtri_kernel_lower (char diag,
        const __global double *A, __global double *d_dinvA, uint lda)
{
    int i,j;    double Ystx=0;
    __local double * Bw, *x=NULL, *y=NULL; const __global double *Aoff=NU
    __global double *my_d_dinvA;
        int switcher=0;
    // Thread index
    uint tx = get_local_id(0);
    int txw;
    // Block index
    uint bx = get_group_id(0);
    Aoff = A+bx*lda*BLOCK_SIZE+bx*BLOCK_SIZE;
    my_d_dinvA = d_dinvA+bx*BLOCK_SIZE*BLOCK_SIZE;
    __local double workspace[BLOCK_SIZE];
    __local double Bs[BLOCK_SIZE*BLOCK_SIZE];
    #pragma unroll
    for (i=0; i<BLOCK_SIZE; i++)
        Bs[i*BLOCK_SIZE+tx] = ((double)(tx>=i))*(*(Aoff+i*lda+tx));
    barrier(CLK_LOCAL_MEM_FENCE);
    switcher = (diag=='u' || diag=='U');
+-- 19 lines: int diagsw = (Bs[tx*BLOCK_SIZE+tx]==0);-------------------
        y[tx] = (double)switcher*Ystx*(-Bs[i*BLOCK_SIZE+i])+(double)(!swi
        barrier(CLK_LOCAL_MEM_FENCE);
    }
+--  3 lines: #pragma unroll--------------------------------------------
}
```

🟪 Lines with difference
🟥 Different keywords
🟦 Same code segments (folded)

Figure 2: Comparison of Device Kernel Code Between OpenCL and CUDA.

CUDA term	OpenCL term
host CPU	host
streaming multiprocessor (SM)	compute unit (CU)
scalar core	processing element (PE)
host thread	host program
thread	work-item
thread block	work-group
grid	NDRange
shared memory	local memory
constant memory space	constant memory
texture memory space	constant memory

Table 1: Comparison of terms used by CUDA and OpenCL to describe very similar concepts.

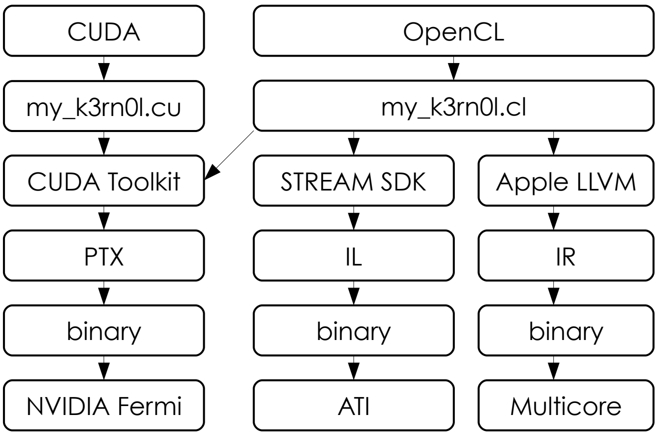


Figure 1: Comparison of software stacks used with CUDA and OpenCL on various hardware platforms.

3.1. *Relation to CUDA*

Figure 2 shows side-by-side differences of the kernel codes for triangular inversion routine (TRTRI) for OpenCL and CUDA. The changes are in the lines annotated in red. They belong to the following categories:

- Obtaining the ID for the thread/work-item and block/work-group.
- The definition of shared memory in CUDA is replaced in OpenCL by local memory: __shared__ is replaced with __local
- OpenCL makes explicit differentiation between global memory addresses (device memory address space) and local memory addresses (register variable or pointer to shared memory) whereas CUDA makes no such distinction.
- Syntax for synchronization primitives.

Differences in the CUDA and OpenCL front-ends yield different timing profiles.

3.2. *Profiling*

Unlike CUDA, OpenCL requires environmental setup on the host (CPU) before launching kernels to run on GPU. This process also includes compiling kernels. The process for setting up kernels to execute is similar for all GPU kernels and an analysis of TRSM gives a typical breakdown of initialization. Figure 3 shows the breakdown of initialization time. We run oclDtrsm (OpenCL double precision triangular solver) with

ATI term	NVIDIA term
Streaming Multiprocessor	Stream Core
streaming multiprocessor (SM)	compute unit (CU)
scalar core	processing element (PE)
shared memory	shared memory
texture cache	texture cache
PTX	IL
binary	binary

Table 2: Comparison of terms used by ATI and NVIDIA to describe very similar concepts.

M=10240 and NRHS=128 on an Intel Q9300 running at 2.5 GHz and a Tesla C2050. Setting up the kernel takes longer than the kernel execution itself.

Compiling OpenCL source code into an intermediate representation takes the most time in initialization. We observed similar results on older NVIDIA cards (e.g. GTX 280) and ATI cards (e.g. Radeon 5870) using ATI STREAM SDK[25]. On the Tesla C2050, the compilation of 300+ lines of OpenCL C code into 7000+ lines of PTX takes just over 2 seconds, while the computation on fairly large problem takes less than 0.2 second. This overhead can lead to a severe performance impact if not accounted for when dealing with many OpenCL routines calling each other in a software library. One solution to reduce this overhead is to separate compilation and execution.

Since OpenCL includes separate compilation and build functions in its API, source code compilation can be performed once during the deployment/installation stage of the math library. As of writing, there is no off-line kernel compiler in NVIDIA's OpenCL platform. Documentation suggests [26] for this to be implemented by the developers. We can do this by fetching the Intermediate Representation (IR) resulting from compilation using clGetProgramInfo and saving it to disk. During the initialization phase, the IR can be read from disk and processed with a call to `clBuildProgram`. This method reduced the time of getting the binary code ready to run from 2+ seconds to 0.2 seconds. While the time to create a kernel from a pre-built program still takes more time than a single TRSM, initialization is 10x faster when the raw source code isn't compiled every time the user runs the application. Having sped up initialization, the time profile for the TRSM kernel itself is the next item to optimize.

GPU Device	NVIDIA GTX 280	NVIDIA C2050 (Fermi)	ATI Radeon 5870
Compute Units	30	32	20
Processing elements	8	16	16

Table 3: Comparison of computational resources available on NVIDIA's GTX 280

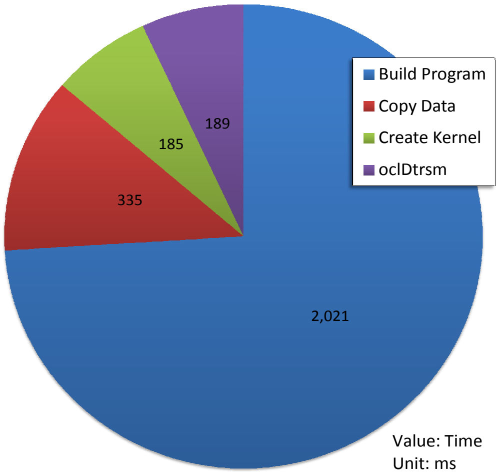


Figure 3: Runtime break down

The performance of TRSM is dominated by GEMM[27]. Since GEMM is one of the most important kernels in linear algebra, we will focus on implementing and analyzing a fast OpenCL GEMM in the coming sections.

4. NVIDIA Platform

4.1. Fermi Architecture

Fermi is NVIDIA's latest GPU product line that includes the GTX4xx series and the Tesla C2050. It introduces several changes over the previous GPU offerings including more plentiful and capable compute units and a revamped memory hierarchy. Under Fermi, more compute units are available, warp sizes have increased from 16 to 32 threads, and each compute unit issues instructions from 2 warps concurrently [28]. Multiple kernels can run simultaneously on Fermi hardware as opposed to previous generations which only support a single kernel executing at any given time [7]. This feature increases device utilization for matrix operations with small problem sizes by increasing the number of thread blocks beyond that which a single kernel allows. On the Tesla C2050 GPU, the number of double precision ALUs as compared to single precision ALUs has increased to 1:2; for every double precision ALU, there are 2 single precision ALUs [29]. In previous generations, this ratio was 1:8. This implies the double precision peak performance (515 GFlops/s [30]) is half that of single precision (1.015 TFlops/s [30]). In addition to extra compute units, Fermi revamps the memory architecture of previous generation GPUs.

Fermi's retooled memory system mainly features changes in caching. Global memory loads and stores are now fetched through the L1 cache, which shares hardware with shared memory. Shared memory and L1 can be configured as 48kB/16kB or 16kB/48kB respectively. The former configuration can increase occupancy in applications that use a large amount of shared memory while the latter configuration can decrease global memory access latencies within a compute unit. Fermi also increases

Figure 4: NVIDIA C2050 (Fermi) architecture.

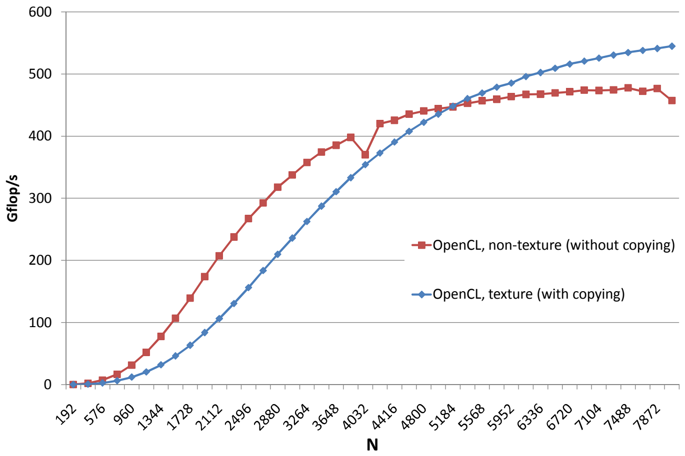


Figure 5: MAGMA SGEMM with/without texture (copying) in OpenCL

the number of registers (the other limiting factor in occupancy) available to applications to 128kB per compute unit[29]. The Tesla C2050 has 144GB/s of memory bandwidth.

4.2. *Fast GEMM*

We have previously published a fast GEMM algorithm for the Fermi architecture [31]. This work expanded on the work of Volkov and Demmel [14], who provided a high performance matrix multiply for NVIDIA's previous generation GPUs. The new algorithm is available in the MAGMA library and will be included in CUBLAS 3.2. Both MAGMA and Volkov's GEMM algorithms are written in CUDA. In this work, we rewrite the MAGMA algorithm in OpenCL, tune the implementation for the Fermi architecture, and compare the OpenCL implementation performance to that of CUDA. Additionally, we run the MAGMA algorithm on both NVIDIA and ATI GPUs, illustrating OpenCL's cross platform design and examining the portability of algorithm performance.

4.3. *From CUDA to OpenCL*

Given the performance of MAGMA GEMM on Fermi, the intuitive fast OpenCL implementation is translated from the CUDA. While CUDA offers the ability to bind a texture to global memory for direct access, which is used in MAGMA GEMM, OpenCL does not. To access matrices through the texture cache, data in global memory must be explicitly copied into an image before it can be read by GPU kernels. Figure 5 shows the performance of 2 SGEMM kernels - one using images (requiring a copies) and one that directly accesses global memory.

Our kernels for copying global buffers into images don't run efficiently on Fermi. We show the memory bandwidth utilization when copying an NxN matrix into an image in Figure 7. These kernels use less than 7% of the Tesla C2050's peak bandwidth. We found that using clEnqueueCopyBufferToImage improves performance fourfold. For our SGEMM using the texture cache, we need to copy without transposition both A and B into images with a single floating point alpha channel. For DGEMM we copy A and B into images with red and green single precision channels, the concatnation of which represents one double precision number in the matrix. High performance data copies are important for small matrix multiplications where there is less work to amortize them.

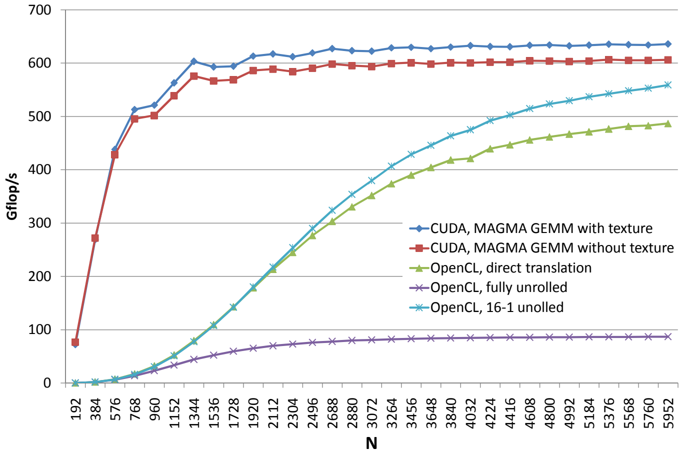


Figure 6: MAGMA SGEMM with CUDA and OpenCL

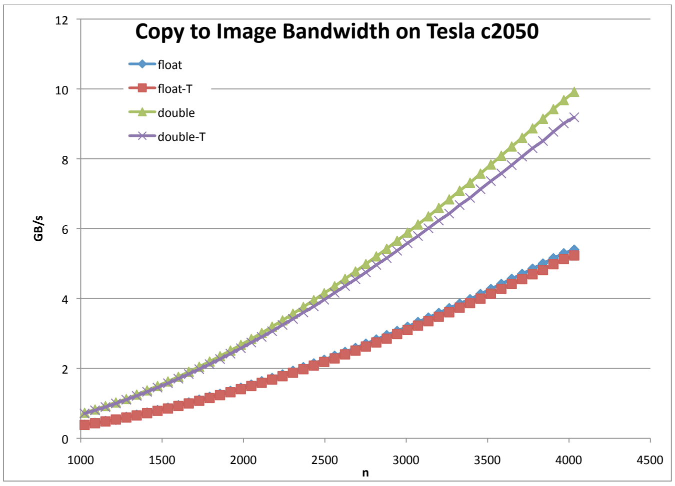


Figure 7: Bandwidth when copying from a global buffer to an image for float and double types with and without transposition

Figure 6 shows the CUDA performance with and without using texture in MAMGA's Fermi GEMM. We show that not using the texture cache leads to a 5% performance drop, which means that streaming data through textures on Fermi contributes little performance. This leads to the idea of removing image copies to save time, which counteracts the performance drop by not using textures. Figure 5 demonstrates this by also showing the performance of the texture-free version of the same code. The cross over point shows that copying overhead is expensive and only pays off for larger problem size.

The way OpenCL code is converted from CUDA is straightforward. Firstly, texture binding and fetching are removed from the MAGMA's Fermi GEMM in CUDA. Then the code is trans-

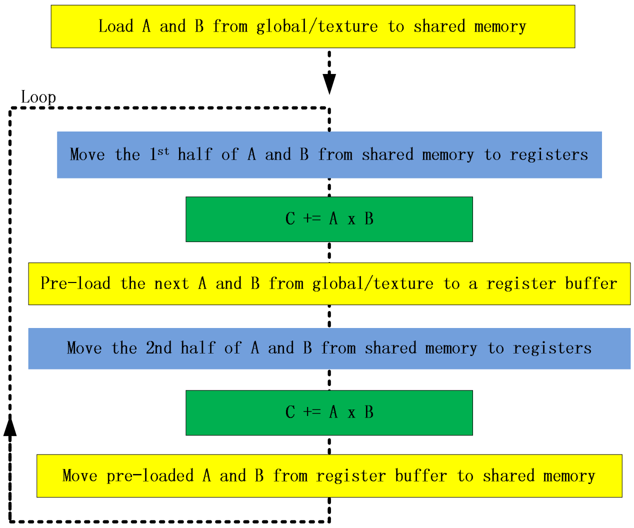


Figure 8: MAGMA GEMM structure

lated into OpenCL using keyword changes previously described. Most of the data I/O and computation instructions remain the same. The green line in figure 6 shows that the performance of this code ramps up very slowly and drops slightly more than 100 Gflop/s at large matrix size. We examing the PTX output from OpenCL and CUDA To explain the different performance.

The kernel structure of MAGMA's Fermi GEMM is shown in figure 8. On the NVIDIA platform, both CUDA and OpenCL produce PTX, NVIDIA's intermediate representation between high-level code and device-specific assembly code. In the three yellow parts, ld.global.f32 is issued to load data from global memory, and in the blue parts, ld.shared.f32 is issued to load data from shared memory. When both operands A and B are in registers, FMA is issued to calculate $C+=A*B$.

By comparing the PTX generated by CUDA and OpenCL compiler, the first major difference is that the CUDA compiler generates code that reads data from shared memory right before the computation that requires it. For example, the first blue and green boxes have the following code:

```
#pragma unroll
for( int y=0;y<6;y++)
        Bxp[y]= Bb[j1][res+y*16];
#pragma unroll
for( int y=0;y<6;y++)
        Axs[y]  =  Abs[qot+y*16][j1]  ;
#pragma unroll
for( int x=0;x<6;x++) {
        #pragma unroll
        for( int y=0; y<6; y++)
                Cb[x*6+y] += Axs[x]*Bxp[y];
}
```

The CUDA compiler's resultant output is:

```
ld.shared.f32    %f508, [%rd53+3276];
mov.f32     %f509, %f391;
fma.rn.f32  %f510, %f508, %f446, %f509;
mov.f32     %f511, %f510;
mov.f32     %f512, %f394;
fma.rn.f32  %f513, %f508, %f450, %f512;
mov.f32     %f514, %f513;
mov.f32     %f515, %f397;
......
......
fma.rn.f32  %f525, %f508, %f466, %f524;
mov.f32     %f526, %f525;
```

Here shared memory data at address %rd53+3276 is fetched to register %f508 and 6 FMAs are issued to consume this value. This pattern allows better overlap of I/O and computation and therefore the overhead of slow I/O can be better hidden through pipelining.

OpenCL compiler emits different PTX. There following is the equivalent PTX of the above C code:

```
ld.shared.f32    %f315, [%r120+1088];
ld.shared.f32    %f316, [%r120+2176];
......
......
ld.shared.f32    %f326, [%r184];
add.s32     %r184, %r184, 388;
fma.rn.f32  %f375, %f319, %f325, %f375;
fma.rn.f32  %f376, %f319, %f324, %f376;
......
......
fma.rn.f32  %f386, %f318, %f326, %f386;
```

This PTX appears to follow the C code directly: all the loading from shared memory followed by all FMAs. While executing in a loop, this pattern could stall GPU cores waiting for all the data to load from shared memory.

Another PTX discrepancy is that the OpenCL compiler does not unroll the outermost loop despite the unroll pragma. In order to make a fair comparison, we manually unrolled all the inner loops so that unrolling pragma works on the outermost loop. The performance of this code is shown in figure 6 as 'fully unrolled'. After seeing drastic drops in performance, we found that OpenCL compiler unrolled the outermost loop and grouped different iterations' ld.shared.f32 followed by many FMA operations. Similar results have been observed on the ATI's OpenCL compiler. This grouping action further exacerbates stalls and explains the low performance. To demonstrate this claim, different combinations of manual unrolling and pragma locations were tested to fight with the OpenCL compiler's tendency of grouping similar operations. The best solution we found is putting the two blue and green parts together forming a 16-loop rather than two 8-loop, which is unrolled using pragma with a factor of 15 (hence the name '16-1 unrolling'). This incomplete unrolling tricks the compiler into laying out the codes in small groups of ld.shared.f32 and ld.shared.f32. This lifts the performance to within 50 Gflop/s of the original CUDA code as shown in figure 6. If the compiler better collected the ld.shared.f32 and ld.shared.f32 instructions into small groups interleaved with execution, the performance

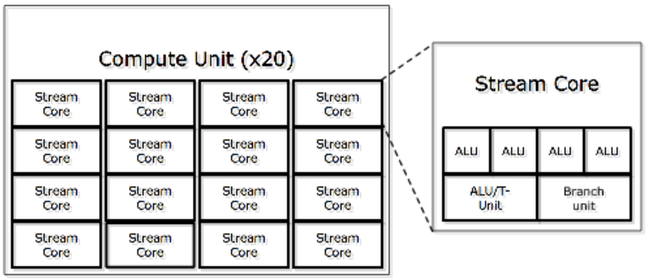


Figure 9: Radeon 5870 architecture

of the OpenCL code for this and similar operations should closely match the CUDA performance. Our ultimate objective is to elevate the performance of the sans-texture kernel to the CUDA non-texture kernel, and using the texture-based kernel for large problems where the copy costs can be amortized.

5. ATI Platform

5.1. Radeon 5870 Architecture

ATI's Evergreen architecture has many analogues with NVIDIA GPUs. Like Fermi-based cards, the Radeon 5xxx GPUs have copious memory bandwidth and many parallel computation units to exploit data parallelism. Our work focuses on the Radeon 5870.

The Radeon 5870 GPU has 1600 ALUs organized in groups (fig. 9). This graphics processor has 20 compute units, each of which contains 16 stream cores. Each stream core within a compute unit executes an instance of a kernel in lockstep. This SIMD hierarchy is analogous to NVIDIA's SIMT engines. However, unlike NVIDIA's Fermi architecture, each stream core is a 5 ALU Very Long Instruction Word (VLIW) processor. Threads are grouped and interleaved on compute units.

Threads are interleaved to hide memory access latency. They are grouped into sets of 64 called a wavefront. A wavefront is analogous to a warp on NVIDIA hardware. Of these 64 threads, 16 execute on a given clock cycle on the 16 stream cores within a compute unit. Over the course of 4 clock cycles, all threads are interleaved, executing an instruction. This hides memory latency; while one thread is loading data another thread can be performing computation[32].

Each ALU in a stream core can independently perform basic floating point operations. In single precision, each ALU can issue one basic floating point instruction such as subtract, multiply, Multiply-Add (MAD), etc. instruction per clock cycle with a pipeline latency of 8 cycles. In addition to basic floating point instructions (such as multiply, add, subtract, divide, and MAD), the fifth ALU can perform transcendental functions including log, square root, etc. To perform double precision floating point operations, some number of the ALUs are fused together. In the case of Fused Multiply Add (FMA), four of the five ALUs are used per operation[32]. The fifth unit is free to do integer or single precision calculations during this time. Since four ALUs are fused per operation and the fifth does not perform double precision operations, the peak double precision throughput is 1/5 that of single precision. The Radeon 5870's 1600 ALUs run at 850MHz, yielding a peak throughput of 2.72TFlops/s in single precision and 544GFlops/s in double. Like NVIDIA's offerings, the Radeon 5870 has high off-chip memory bandwidth and even higher on-chip bandwidth.

To balance the performance of floating point units, the Radeon 5870 features high bandwidth to global memory augmented with a texture cache. Global memory has a peak data throughput of 154GB/s divided over 8 memory controllers. Unlike the Fermi architecture, reads and writes to global memory are generally not cached. However, reads and writes to textures are cached. Each compute unit has its own 8KB L1 cache yielding an aggregate bandwidth of 1TB/s and can produce 4 bytes of data (1 float or half a double) per cycle per stream core. Multiple compute units share a 512kB L2 cache with 435GB/s of bandwidth between L1 and L2. In addition to automatically controlled texture caches, data reuse can also be facilitated using shared memory[32].

The Radeon 5870 features 32KB of shared memory per compute unit. This shared memory has provides 2TB/s of aggregate bandwidth. As with the Fermi architecture, local memory usage dictates how many concurrent wavefronts can run on a compute unit. Each compute unit can produce 2 4-byte (2 floats or 1 double) shared memory requests per cycle. As with NVIDIA cards, bank conflicts can hurt shared memory performance and need to be minimized. Since each stream core can perform 5 MADs per cycle in single precision, more bandwidth is needed for operands than shared memory can provide. This is also true for double precision. As such, register blocking becomes crucial in obtaining a high performance GEMM kernel. [32]

Registers provide the highest memory bandwidth on-chip. The Radeon 5870 has 256KB of register space per compute unit (5.1MB for the whole GPU) and can produce 48 bytes/cycle of data per stream core. Results generated on the previous cycle used as operands don't count towards this limit, as they can be forwarded using the Previous Vector or Previous Scalar register [32]. Each of the 5 MADs per cycle takes 4 operands yielding 20 total operands of 4 bytes a piece. 4 of these operands can be mitigated using the previous vector register and one of these operands can be shared among the 5 ALUs. This equates to exactly 48 bytes/cycle needed to fully utilize all 5 ALUs in a perfectly scheduled SGEMM. If the scheduling is not perfect, registers can actually serve as a bottleneck in the computation. For DGEMM, registers provide sufficient bandwidth for the single FMA instruction per stream core regardless of operand reuse. Like shared memory, register usage dictates the number of concurrent wavefronts executing on a compute unit. Unlike NVIDIA GPUs, registers are 128-bit and support swizzling at no performance cost.

5.2. Fast GEMM

We present a fast GEMM algorithms for single and double precision optimized for ATI hardware. These kernels are based on Nakasato's matrix multiply written in ATI's Intermediate Language (IL)[33]. This work focused on computing row

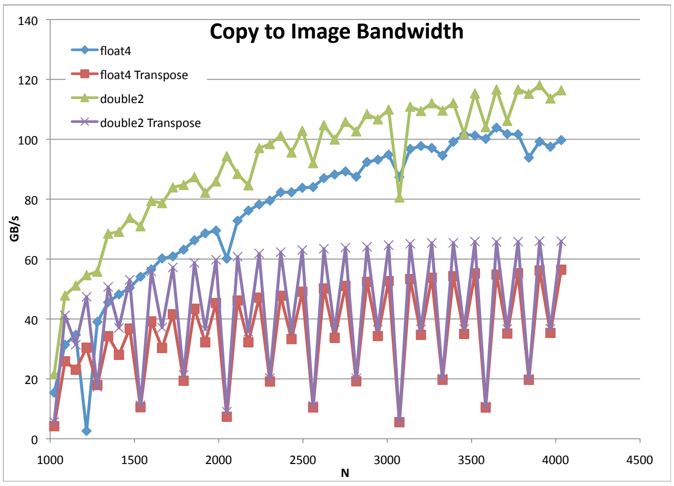


Figure 10: ATI Memory bandwidth utilization during image copies

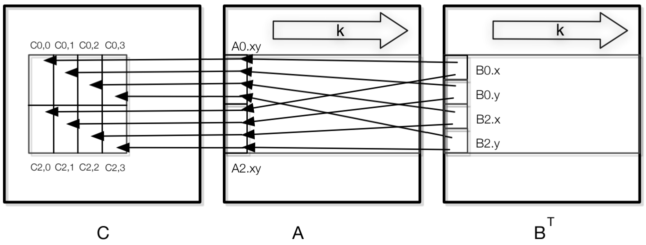


Figure 11: ATI DGEMM algorithm

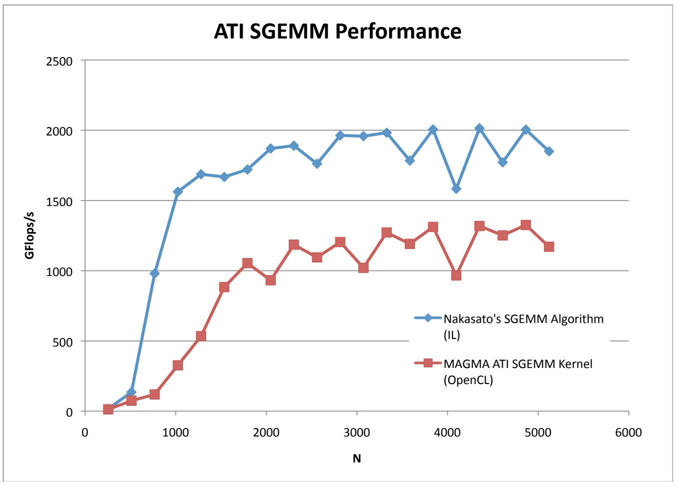


Figure 12: ATI SGEMM performance

major matrix multiplication $C = A^T B$. We expanded this work to perform a column major $C = \alpha AB^T + \beta C$. Our algorithm makes extensive use of the texture cache, which requires copies in OpenCL. We use custom copy kernels to move data into images configured with RGBA color mode and float values. We pad the leading dimension of the image with zeros if needed. For single and double precision, we pad the leading dimension of the image to a multiple of 4 float and 2 doubles respectively. In double precision, the concatenation of the red and green channels represents the first double, while the blue and alpha channel represent the second double. A is copied without transposition into its corresponding padded image while B is transposed and copied into its image. The time to perform these copies is $O(N^2)$, which is amortized by the $O(N^3)$ operations in GEMM for large problems. Copying A into an image efficiently uses the Radeon 5870's memory bandwidth (fig. 10), while copying B doesn't.

The kernels copying data from global memory into images achieve a high fraction of the Radeon 5870's available bandwidth. For non transposed copies, our kernels used over 100GB/s of the 154GB/s of available bandwidth with little variance. Our transpose-and-copy kernels achieved half that amount and were far more sensitive to problem size. Poor memory coalescing in the transpose kernels is to blame for the low memory throughput. These kernels and the ones needed for texture use for the NVIDIA texture-based kernel ran an order of magnitude more quickly on the Radeon 5870 than on the Tesla C2050. Oddly enough, our float and double copy kernels were significantly faster than clEnqueueCopyBufferToImage on the Radeon 5870.

Once A and B have been copied, the matrix multiplication kernel executes. Our DGEMM algorithm is shown in fig. 11. Each thread computes a single 4x4 block of C as double2s. Since B is transposed, the columns of B reside in its leading dimension. The rows of A reside in its leading dimension. We scale the double2 row vectors of A by the corresponding double of B using swizzling to extract and duplicate the required element. This scaled vector is then accumulated into the corresponding double2 of C. All of this is done with a single MAD and swizzling.

To maximize outstanding loads from the texture cache, we use 8 samplers. 2 samplers load 2 double2s from A into registers and 2 samplers fetch 2 double2s of B. We unroll the k loop twice to use the other 4 samplers. A useful feature of images is that when its sampler is declared using CL_CLAMP, fetching outside the bounds of the 2D image yields 0.0, which has no effect on the result when accumulated. This allows k to be indivisible by 2 yet still function correctly; in this case, our DGEMM kernel samples beyond the bounds of A and B and accumulates 0.0 into the result. Handling cases when m is not a multiple of 2 is non-trivial and our algorithm currently doesn't handle this case. In fact, our algorithm requires both m and n be multiples of 4. Padding C can overcome this limitation. SGEMM is analogous to our DGEMM kernel, where each thread computes an 8x8 block of C in float4 registers and has analogous limitations.

We compare our OpenCL results to Nakasato's IL performance in 12 and 13. Nakasato's timings do not include copy times, which exaggerates performance for small problems. Our timings do include this copy. Furthermore, Nakasato's algorithm performs only the matrix multiplication while we perform the alpha and beta scaling (a negligible amount of time for large N). Neither timings include PCIe data transfer times, which would be amortized in large multiplications or when data can be heavily reused on the GPU.

Our SGEMM kernel exceeds 1.3TFlops/s for N=3840, 4352,

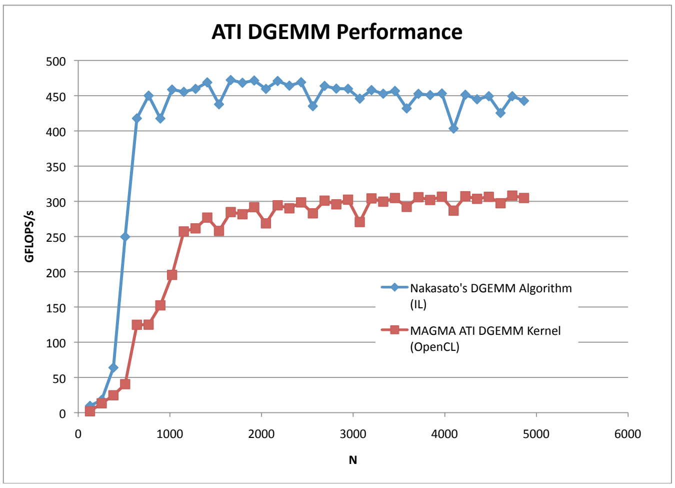


Figure 13: ATI DGEMM performance

and 4864. Nakasato's IL implementation just exceeds 2TFlops/s for these same N, implying our OpenCL MAGMA implementation achieves 65% of a fast matrix multiply algorithm written in high-level assembly code. Nakasato achieves 74% of the Radeon 5870's peak performance while we achieve 49%.

Our OpenCL ATI DGEMM algorithm achieves 308GFlops/s on the Radeon 5870. In comparison, Nakasato's matrix multiply algorithm achieves 472GFlops/s. This means that our OpenCL implementation is has 69% of the performance of Nakasato's IL matrix multiply. The ATI OpenCL kernel computes at 57% of the hardware's peak performance while Nakasato's kernel operates at 87% of maximum throughput. From this performance comparison, we illustrate that OpenCL provides fair performance with a high degree of programmability on ATI hardware. Furthermore, we found that the relevant data copy and pack kernels effectively used the Radeon 5870's memory bandwidth (fig. 10).

6. Performance Portability

In this section, we run our device specific kernels on hardware for which they aren't tuned to evaluate performance portability. OpenCL is designed with program portability in mind. Despite different vendors having added extra functionality in their OpenCL implementations, our work only uses features in the OpenCL standard[12]. This theoretically allows the front-end and GPU kernels to run on any platform without changes.

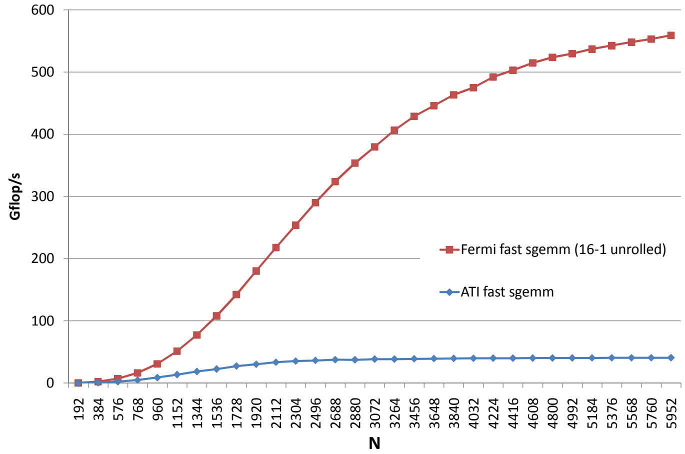


Figure 14: Fast ATI SGEMM running on Fermi card in OpenCL

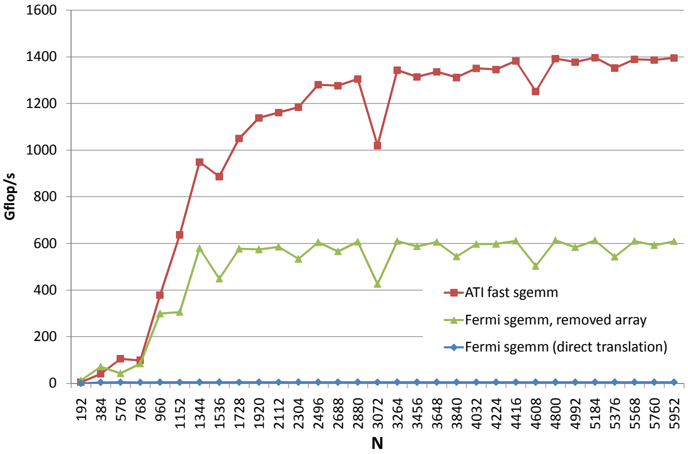


Figure 15: Fast Fermi SGEMM running on ATI Radeon 5870 card in OpenCL

Figure 14 shows our ATI SGEMM kernel running on a Tesla C2050. While achieving 1+ Teraflop/s (50+% peak) on a Radeon 5870, it only manages to execute at 40 Gflop/s(4% peak) on the Tesla C2050. We reverse this experiment in figure 15 and run the OpenCL version of MAGMA's Fermi GEMM on the Radeon 5870. While achieving 400+ Gflop/s (40+% peak) on the Tesla C2050, it has a very low performance when run on a Radeon 5870. Through reading the IL generated, we suspect that despite hinting the arrays that hold operands should be exist in registers, they actually reside in global memory, leading orders of magnitude more data fetch latency. We suspect this is because the compiler fails to fully unroll loops accessing these arrays and as such generates code that performs runtime indexing (which can't occur on registers). To resolve this, the arrays are replaced with standalone variables and performance shoots to 600 Gflop/s (22% peak).

From these two experiments we show that performance is not portable through simple using OpenCL. Without paying attention to the underlying architecture and designing algorithms accordingly, an algorithm's performance suffers.

7. Performance Portability with Auto-tuning

The goal behind the OpenCL standard is to provide functional portability, or in oder words, to enable a single OpenCL application to run across a variety of hardware platforms. Although extremely important, functional portability by itself, e.g., without performance portability, would be insufficient to establish OpenCL in the area of high-performance scientific computing. In this section we address this issue by discussing *auto-tuning* – the vehicle that we recognize as the potential driver of OpenCL applications towards performance portability.

Automatic performance tuning (optimization), or auto-tuning in short, is a technique that has been used intensively on CPUs to automatically generate near-optimal numerical libraries. For example, ATLAS [19, 20] and PHiPAC [21] are used to generate highly optimized BLAS. The main approach for doing auto-tuning is based on empirical optimization techniques. Namely, these are techniques to generate a large number of parametrized code variants for a given algorithm and run these variants on a given platform to discover the one that gives the best performance. The effectiveness of empirical optimization depends on the chosen parameters to optimize, and the search heuristic used. A disadvantage of empirical optimization is the time cost of searching for the best code variant, which is usually proportional to the number of variants generated and evaluated. Therefore, a natural idea is to combine it with some "model-driven" approach in a first stage that would limit the search space for the second stage of an empirical search.

Work on auto-tuning CUDA kernels for NVIDIA GPUs [22, 23] has already shown that the technique is a very practical solution to easily port existing algorithmic solutions on quickly evolving GPU architectures and to substantially speed up even hand-tuned kernels. We expand this early work, as described below, in the context of todays high-end GPGPU from NVIDIA and ATI, using both CUDA and OpenCL.

7.1. Auto-tuning Infrastructure

The performance of CUDA GEMM implementations rely on a number of very well selected parameters and optimizations [18]. Previous work in the area has managed to auto-tune the selection of these parameters and optimizations used, to quickly find the best performing implementations for particular cases of GEMM [22, 23]. However, with the introduction of the Fermi architecture, these auto-tuning frameworks were not able to find the new "optimal" implementations for Fermi, simply because their search space did not consider the newly introduced features in the architecture [31]. Performance portability problems are even further aggravated when porting kernels across hardware vendors – kernels optimized for one vendor's architecture perform poorly on another vendor's architecture, e.g., as illustrated throughout the paper with the GEMM kernels optimized correspondingly for NVIDIA and ATI GPUs. Therefore, our work on providing performance portability has concentrated on building up an auto-tuning infrastructure with the following two-components (characteristic for a complete auto-tuning system):

Code generator The code generator produces code variants according to a set of pre-defined, parametrized templates and/or algorithms. Currently we have identified and collected best GEMM candidates for both NVIDIA and ATI GPUs. We have identified several key parameters that affect performance. The code generator will automatically create kernels using parameters and applying certain state of the art optimization techniques.

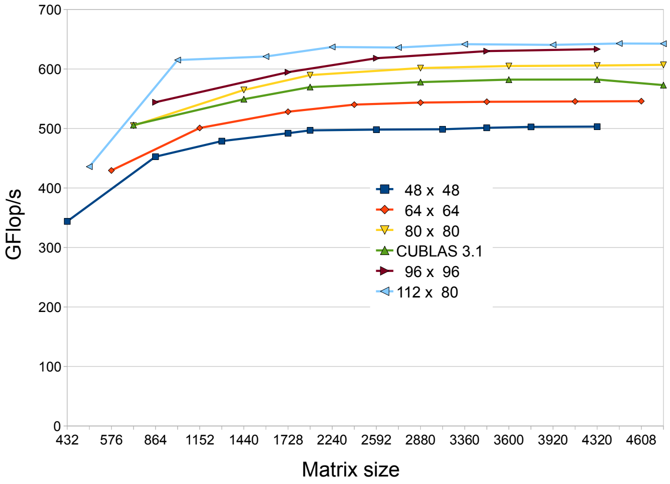


Figure 16: Performance of various automatedly generated SGEMM kernels for Fermi C2050 GPU.

Heuristic search engine The heuristic search engine runs the variants produced by the code generator and discovers the best one using a feedback loop, e.g., the performance results of previously evaluated variants are used as a guidance for the search on currently unevaluated variants.

7.2. Tuning GEMM

To illustrate the effect of auto-tuning we present numerical results in tuning SGEMM. In particular, we parametrize the new Fermi GEMM kernels [31]. Figure 16 gives the single precision performance of several versions derived from the original and run on a Fermi C2050 GPU. The parametrization is based on the size of the submatrix computed by a thread block.

8. Performance Results on Multicore

So far we have presented performance results for GPU platforms from ATI and NVIDIA that we were able to obtain using CUDA and OpenCL software stacks. To put these numbers in a proper perspective, it is informative to compare them to the results obtained from optimized BLAS routines coming from ATLAS and Intel's MKL (Math Kernel Library) running on a multicore hardware – a well researched experimental setting. We use ATLAS as a representative of performance levels achievable from C with extensive use of auto-tuning (the tuning of ATLAS library may take many hours). Intel's MKL represents one of the best performing libraries for Intel processors. When such libraries are distributed by the hardware vendor, most performance sensitive kernels written in optimized assembly code. The results from Intel Tigerton multicore system are shown in Figure 17. The tested system had 4 processors, each one being a quad-core clocked at 2.4 GHz for the total peak performance of 307.2 Gflop/s in single precision and half of that in double precision.

The summary of the results is presented in Table 4 in relative terms (as the percentage of the peak performance) to allow easy comparison of the tested platforms regardless of their absolute and achieved performance. A common theme may

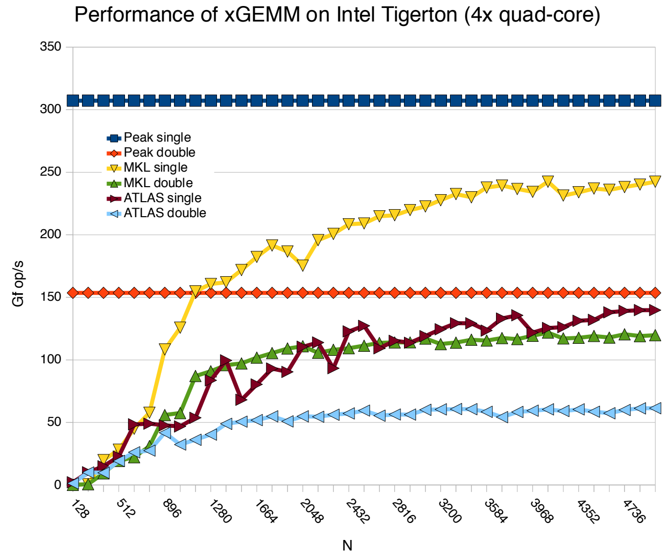


Figure 17: Performance of xGEMM routines on a multicore processor with 16 cores.

	Single precision	Double precision
ATI OpenCL	52%	57%
ATI IL	74%	87%
NVIDIA CUDA	63%	58%
NVIDIA OpenCL	57%	49%
Multicore (vendor)	79%	79%
Multicore (auto-tuned)	46%	40%

Table 4: Comparison of efficiency achieved on the tested hardware.

be observed from this summary: computational kernels written in low-level langauge (such as ATI's IL and x86 assembly) achieve around 80% of peak performance while high-level languages achieve about 50% of peak.

9. Conclusions

In this paper, we evaluated various aspects of using OpenCL as a performance-portable method for GPGPU application development. Profiling results show that environment setup overhead is large and should be minimized. Performance results for both the Tesla C2050 and Radeon 5870 show that OpenCL has good potential to be used to implement high performance kernels so long as architectural specifics are taken into account in the algorithm design. Even though good performance should not be expected from blindly running algorithms on a new platform, auto-tuning heuristics can help improving performance on a single platform. Putting these factors together, we conclud that OpenCL is a good choice for delivering a performance-portable application for multiple GPGPU platforms.

References

[1] E. S. Larsen, D. McAllister, Fast matrix multiplies using graphics hardware, in: Proceedings of Supercomputing 2001, Denver, CO, 2001.

[2] Ádám Moravánszky, Dense matrix algebra on the GPU, available at: `http://www.shaderx2.com/shaderx.PDF` (2003).

[3] J. Krüger, R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, ACM Transactions on Graphics 22 (2003) 908–916.

[4] J. D. Hall, N. A. Carr, J. C. Hart, Cache and bandwidth aware matrix multiplication on the GPU, Tech. Rep. UIUCDCS-R-2003-2328, UIUC (2003).

[5] K. Fatahalian, J. Sugerman, P. Hanrahan, Understanding the efficiency of GPU algorithms for matrix-matrix multiplication, in: In Graphics Hardware 2004, 2004, pp. 133–137.

[6] N. Galoppo, N. K. Govindaraju, M. Henson, D. Manocha, LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware, in: Proceedings of Supercomputing 2005, Seattle, WA, 2005.

[7] NVIDIA, NVIDIA CUDA^{TM} Programming Guide Version 3.0, NVIDIA Corporation, 2010.

[8] NVIDIA, NVIDIA CUDA^{TM} Best Practices Guide Version 3.0, NVIDIA Corporation, 2010.

[9] S. Tomov, R. Nath, P. Du, J. Dongarra, MAGMA version 0.2 User Guide, http://icl.cs.utk.edu/magma (11/2009).

[10] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. C. Sorensen, LAPACK User's Guide, Third Edition, Society for Industrial and Applied Mathematics, Philadelphia, 1999.

[11] R. J. Rost, D. Ginsburg, B. Licea-Kane, J. M. Kessenich, B. Lichtenbelt, H. Malan, M. Weiblen, OpenGL Shading Language, 3rd Edition, Addison-Wesley Professional, 2009.

[12] A. Munshi (Ed.), The OpenCL Specification, Khronos OpenCL Working Group, 2009, version: 1.0, Document Revision:48.

[13] M. Papadopoulou, M. Sadooghi-Alvandi, H. Wong, Micro-benchmarking the GT200 GPU, Tech. rep., Computer Group, ECE, University of Toronto (2009).

[14] V. Volkov, J. Demmel, Benchmarking GPUs to tune dense linear algebra, in: Supercomputing 08, IEEE, 2008.

[15] NVIDIA, NVIDIA Compute PTX: Parallel Thread Execution, 1st Edition, NVIDIA Corporation, Santa Clara, California, 2008.

[16] A. Kerr, G. Diamos, S. Yalamanchili, A characterization and analysis of ptx kernels, in: Proceedings of 2009 IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 3–12.

[17] J. Stratton, S. Stone, W. mei Hwu, MCUDA: An efficient implementation of CUDA kernels on multi-cores, Tech. Rep. IMPACT-08-01, University of Illinois at Urbana-Champaign, `http://www.gigascale.org/pubs/1278.html` (Mar. 2008).

[18] M. Wolfe, Special-purpose hardware and algorithms for accelerating dense linear algebra, HPC Wire`http://www.hpcwire.com/features/33607434.html`.

[19] R. C. Whaley, A. Petitet, J. J. Dongarra, Automated empirical optimization of software and the ATLAS project, Parallel Computing 27 (1–2) (2001) 3–35.

[20] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, K. Yelick, Self-adapting linear algebra algorithms and software, Proc. IEEE 93 (2) (2005) 293–312. doi:http://dx.doi.org/10.1109/JPROC.2004.840848.

[21] J. Bilmes, K. Asanovic, C.-W. Chin, J. Demmel, Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology, in: ICS '97: Proceedings of the 11th international conference on Supercomputing, ACM, New York, NY, USA, 1997, pp. 340–347. doi:http://doi.acm.org/10.1145/263580.263662.

[22] Y. Li, J. Dongarra, S. Tomov, A note on auto-tuning GEMM for GPUs, in: Computational Science ICCS 2009, Vol. 5544 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009, pp. 884–892.

[23] R. Nath, S. Tomov, J. J. Dongarra, Accelerating GPU kernels for dense linear algebra, in: Proceedings of VECPAR'10, Berkeley, CA, 2010.

[24] R. Weber, A. Gothandaraman, R. J. Hinde, G. D. Peterson, Comparing hardware accelerators in scientific applications: A case study, IEEE Transactions on Parallel and Distributed Systems 99 (RapidPosts). doi:http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.125.

[25] ATI, ATI Stream Software Development Kit (SDK) v2.1, available at: `http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx` (2010).

[26] NVIDIA OpenCL JumpStart Guide: Technical Brief, version 0.9 (April

2009).

[27] P. Du, P. Luszczek, S. Tomov, J. Dongarra, Mixed-tool performance analysis on hybrid multicore architectures, in: Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010), San Diego, CA, 2010.

[28] Tuning CUDA Applications for Fermi Version 1.0, available at: `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_FermiTuningGuide.pdf` (Feb. 2010).

[29] NVIDIA, Nvidia's next generation cuda compute architecture: Fermi v1.1, Tech. rep., NVIDIA Corporation (2009).

[30] NVIDIA, Tesla c2050/c2070 GPU Computing Processor: Supercomputing at 1/10 the Cost, available at: `http://www.microway.com/pdfs/Tesla_C2050_C2070_Final_lores.pdf` (2010).

[31] R. Nath, S. Tomov, J. Dongarra, An improved magma gemm for fermi gpus., Tech. Rep. 227, LAPACK Working Note (July 2010).
URL `http://www.netlib.org/lapack/lawnspdf/lawn227.pdf`

[32] ATI, ATI Stream Computing OpenCL Programming Guide, available at: `http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf` (June 2010).

[33] N. Nakasato, Matrix Multiply on GPU, `http://galaxy.u-aizu.ac.jp/trac/note/wiki/MatrixMultiply`.