Dealing with performance/portability and performance/accuracy trade-offs in heterogeneous computing systems: A case study with matrix multiplication modulo primes

Matthew Wezowicz, B. David Saunder, and Michela Taufer

Computer and Information Sciences, University of Delaware

ABSTRACT

We present the study of two important trade-offs in heterogeneous systems (i.e., between performance versus portability and between performance and accuracy) for a relevant linear algebra problem, matrix multiplication modulo primes. Integer matrix linear algebra methods rely heavily on matrix multiplication modulo primes. Double precision is necessary for exact representation of sufficiently many primes. We examine the performance losses due to the use of OpenCL versus CUDA and the use of double versus single precision. Our results indicate that performance losses from the former are minimal with the benefit of cross-platform portability and from the latter are acceptable when double precision is required.

Keywords: GPU programming, CUDA, OpenCL, linear algebra, matrix multiplication modulo primes

1. INTRODUCTION

The benefits of using heterogeneous computing systems for high-performance computing are recognized by the scientific community. Significant performance gains can be achieved by using GPU's for numerous scientific applications. Still scientists have to deal with two key trade-offs, i.e., trade-offs between performance versus portability and between performance and accuracy. The choice of parallel programming language, CUDA or OpenCL, can impact the performance versus portability trade-off. CUDA supports greater performance on NVIDIA GPU's but is limited to running on this vendor's GPU's. OpenCL is cross-platform but not as highly developed and thus suffers in the performance aspect. The choice of single or double precision for computations can impact the performance versus accuracy trade-off. GPU's have a significant advantage over CPU's in single precision performance but a smaller advantage when it comes to double precision and not all GPU's have double precision support. However a large number of computing tasks require the accuracy provided by double precision.

In this work, we present the study of these two important trade-offs for a relevant linear algebra problem, the matrix multiplication modulo primes. Several important scientific and engineering applications depend on higher level linear algebra tools such as the matrix multiplication modulo primes. Traditionally when executed on CPUs, double precision is necessary for exact representation of sufficiently many primes. We examine the performance losses due to the use of OpenCL versus CUDA and the use of double versus single precision. Our results indicate that performance losses from the former are minimal with the benefit of cross-platform portability and from the latter are acceptable when double precision is required.

The rest of this paper is organized as follows: Section 2 presents a short overview of the matrix multiplication modulo primes; Section 3 discusses the relevance of the mathematical method for scientific applications; Section 4 briefly provides an overview of GPUs, OpenCL and CUDA; Section 5 describes our implementation of the matrix multiplication modulo primes in OpenCL and CUDA; Section 6 discuss the trade-off between performance and portability as well as performance and accuracy for our implementation; and Section 7 ends the paper by summarizing the main contributions.

E-mail: mwezz@udel.edu, saunders@udel.edu, taufer@udel.edu

2. MATRIX MULTIPLICATION MODULO PRIMES

The computational kernels of matrix-vector product and matrix-matrix product are fundamental to scientific computation. The matrix on the left may be dense, leading to a regular computation with predictable data flow and operations, or sparse, leading to an irregular computation. Four possible combinations, also called kernel operations, are possible: mv, mm, sv, and sm (m means dense matrix, v means vector, and s means sparse matrix), where mv and mm are regular kernels while sv and sm are irregular kernels. These two regular and two irregular kernels are central to scientific computation. It is well-known that algorithms for matrix multiplication are expensive ($O(n^3)$). Current implementations of exact matrix multiplication modulo run single threaded on CPUs. These implementations employ algorithms such as Strassen's algorithm ($O(n^{2.807})$) to reduce the computational complexity.

Amongst the matrix multiplication algorithms, matrix multiplication modulo primes presents the additional requirement that we carry out matrix operations in a way that preserves exact results (no roundoff error). Exact matrix multiplication modulo a prime performs floating point operations so that all partial result result values are within a range that fits in the floating point mantissa, to prevent accuracy loss. To do so, most of the operations are modular. Before results exceed the exactly representable range, division by the carefully chosen modulus is performed to produce an exact modular residue (remainder) of the desired result. The entries of a matrix product are dot products of a row of the left matrix times a column of the right one. If the modulus is p and the entries are initially in the range $0..p-1$ then a sum of D products can be computed exactly so long as Dp^2 fits in the mantissa. In other words the driving rule is that $Dp^2 \leq 2^{24}$ in floating point and $Dp^2 \leq 2^{53}$ in double precision. One could also consider 32 bit and 64 bit ints, but on the hardware under consideration here, those operations are unavailable or slow.

The applications (see next section) vary in their requirements regarding the modulus p. For some, p is a tiny prime such as three (3) or five (5). For others, the modulus is selectable and there is a trade-off between the size of the prime and the number of iterations the algorithm needs. In this case, if float is sufficiently much faster than double, a smaller prime p may be used. Finally some algorithms require primes $p > 2^{20}$ and double precision must be used.

Because of these trade-offs, We consider three different algorithms for matrix multiplication modulo primes, i.e., the basic modulus algorithm, the delayed modulus algorithm, and the partial delay modulus algorithm.

The *basic modulus algorithm* assumes that the entries of matrices A and B are in the range $0..p-1$; the computed entries are to be in the same range. The modulus p is a positive integer exactly representable in float (p is in range $2..2^{11}$) and we reduce modulo p after each muladd. The algorithm is as follows:

```
void computeBasicModular(float* C, const float* A, const float* B,
                         unsigned int hA, unsigned int wA,
                         unsigned int wB, float p)
{
  for (int i = 0; i < hA; ++i) {
    for (int j = 0; j < wB; ++j) {
      float sum = 0;
      for (int k = 0; k < wA; ++k) {
        float a = A[i * wA + k];
        float b = B[k * wB + j];
        sum += (a * b);
        sum = fmod(sum, p); // main change for basic modulus algorthm
      }
      C[i * wB + j] = sum;
    }
  }
}
```

The *delayed remainder modulus* algorithm assumes that the entries of the matrices A and B are in range $0..p-1$; the computed entries are to be in the same range. As for the basic modulus algorithm, the modulus p is a positive integer exactly representable in float (p is in range $2..2^{11}$). In contrast to the basic modulus algorithm, the division by p is delayed

maximally for the sake of faster computation (`fmod` is relatively expensive). Thus the algorithm requires $wA * p^2 < 2^{23}$. The algorithm is as follows:

```
void computeModDel(float* C, const float* A, const float* B,
                   unsigned int hA, unsigned int wA,
                   unsigned int wB, float p)
{
   for (int i = 0; i < hA; ++i) {
    for (int j = 0; j < wB; ++j) {
      float sum = 0;
      for (int k = 0; k < wA; ++k) {
        float a = A[i * wA + k];
        float b = B[k * wB + j];
        sum += (a * b);
      }
      sum = fmod(sum, p); // main change for the delayed modulus algorthim
      C[i * wB + j] = sum;
    }
  }
}
```

The *partially delayed remainder* modulus algorithm assumes that the entries of the matrices A and B are in range $0..p-1$ and computed entries are to be in the same range. As for the basic modulus algorithm and the delayed remainder modulus algorithm, the modulus p is a positive integer exactly representable in float (p is in range $2..2^{11}$). Let $D = 2^i$ be the delay factor. Contrary to the the other two algorithms, the division by p is performed only every D mul-adds. This works because $D * p^2 < 2^{23}$ given $p < 2^{11}$ and $D = 2^i$ does not exceed a certain range. Larger D may be used if p is correspondingly smaller. For instance, in the case of $p < 2^5, D = 16$. The algorithm for $D = 32$ is as follows:

```
void computeModDel32(float* C, const float* A, const float* B,
                         unsigned int hA, unsigned int wA,
                         unsigned int wB, float p)
{
  for (int i = 0; i < hA; ++i) {
    for (int j = 0; j < wB; ++j) {
      float sum1 = 0;
      for (int k1 = 0; k1 < wA; k1 += 32) {
        float sum0 = 0;
        for (int k0 = 0; k0 < 32; k0++) {
          float a = A[i * wA + k0 + k1];
          float b = B[(k0 + k1) * wB + j];
          sum0 += (a * b);
        }
        sum1 += fmod(sum0, p); // change for partially delayed algorithm
      }
      sum1 = fmod(sum1, p); // change for partially delayed algorithm
      C[i * wB + j] = sum1;
    }
  }
}
```

3. IMPACT ON RELEVANT SCIENTIFIC APPLICATIONS

Exact computation has an important role to play in high performance scientific and engineering computation. Just as systems such as Mathematica and Maple provide an essential counterpoint to the more numerically oriented Matlab, so

exact linear algebra systems are to high end implementations of basic numeric (approximate) linear algebra including BLAS, LAPACK, and the more specialized supercomputing systems. The LinBox library aims to be the core tool for high performance exact linear algebra. It turns out that the most important kernels for exact computation are for computation with integers modulo a prime. A BLAS-like suite of basic modular routines serves a very analogous role to numeric BLAS.

Exact computation is essential to a number of computational problems including cryptography, structural problems such as solution of polynomial systems, topological characterizations including especially homology, and studies of symmetry including the analysis of Lie groups. Exact computation also can play a role when a purely numeric solution is needed but the system is ill conditioned for numeric methods.

In cryptography, exact computation is essential of course both for encryption/decryption and for attacks on cryptosystems as well. Most of this is modular in nature and some of it is linear algebra as well. For example, the numeric sieve approach to factorization (to break RSA) has as a major, sometimes dominant, component the computation of ranks of very large matrices modulo 2.[1,2]

Other problems involve exact integer computation. Homology is often a good topological, qualitative characterization of an object. Consider the characterization of surfaces in any dimension. Many surfaces are suitably approximated by a simplicial decomposition. For two dimensional surfaces this is an oriented triangular mesh. In turn the homology of the simplicial complex is obtained by integer rank and Smith Normal Form (SNF) computation of it's *boundary matrices*, which are defined in terms of the simplices. Most often, applications require the higher dimensional (simplicial complex) or lower dimensional (graph or network) version of the construction of boundary matrices and computation of the corresponding homology and Betti numbers. This all boils down to integer matrix SNF computation. Speeding SNF computation is a primary purpose of the kernels developed in this paper. Some of the areas in which homology has been successfully applied using high performance integer matrix tools include the study of strongly regular graphs,[3,4] update this citation and of other combinatorial objects.[5–7]

Many additional applications exist and/or are under consideration. A SANDIA workshop on applied algebraic geometry was held in August 2009 in Santa Fe.[8] Applications discussed included the modeling of flame and systems for securing stored nuclear materials. In the study of Lie Groups arises a classification of a space of matrices according to the sign pattern of eigenvalues, including positive definiteness (all eigenvalues positive). When a matrix is near singular, numeric methods for positive definiteness are subject to error, yet the matrices are exact (not consisting of approximately measured data) and for this classification the exact result is essential. Exact linear algebra has been successfully applied[9–11] by computing the characteristic polynomial and using Descartes' rule of signs to determine the sign pattern of the roots.

The above mentioned applications depend on linear algebra tools rank modulo 2, integer rank, Smith Normal Form of an integer matrix, and characteristic polynomial. A few other high level linear algebra problems come up in the SNF computation including integer determinant and rational solution to an integer linear system.

4. CONSIDERATIONS ON GPUS, OPENCL, AND CUDA

GPU programming goes back a number of years to a time well before either CUDA or OpenCL. In the beginning the only way to interact with a GPU was with a graphic API such as OpenGL or DirectX. While these could be used they required signification training and forced problems to be described in graphical terms. Everything had to be specified as 1D, 2D, or 3D textures and operations on these textures commonly called shader kernels. There was also the restriction to the primitive data types used in graphics. These include half floats (16bit), 24bit pixel vectors, and other types generally unsuitable to scientific computation without significant jerry rigging. As GPU's became more powerful and ubiquitous several languages were developed to provided a more familiar environment for programmers. The most prominent of these languages was BrookGPU, which as of 2007 is no longer under active development.[12] It was essentially a front-end for graphics API's made using extensions to ANSI C but was a step forward in usability for the average programmer. Another language was ATI's proprietary Close To Metal. It provided a way to access ATI GPU's native instruction set allowing the programmer to bypass the graphics API entirely.

More recent years have seen the rise of truly high level programming languages for interacting with GPU's. First was NVIDIA's proprietary Compute Unified Device Architecture, hence forth referred to as CUDA, in 2007. It consists of a C like programming language for writing the GPU kernels and extensions to ANSI C for controlling the operation of the GPU from the host computer.[13] A CUDA kernel program is a set of instructions that will be run by all of the threads created on the GPU, and the host controls how many of these threads are created, on which device, and what data is operated on.

CUDA has be come relatively popular because it is easy to use and performs well on NVIDIA's GPU's, which are currently considered the best for scientific computing.

OpenCL is the most recent GPU programming language to come into existence, appearing in late 2008. Originally developed by Apple, Inc., the specification is now managed by the Khronos group in cooperation with groups from Apple, Inc., AMD, IBM, Intel, and NVIDIA. It is similar to CUDA in several aspects. It consists of a C like language for writing GPU kernels and a host API for controlling the GPU.[14] The kernel languages are even very similar in their syntax, keywords, and restrictions. The host code in any OpenCL program is standard code that contains function calls for interacting with OpenCL devices. OpenCL has bindings for C, C++ (more specifically a wrapper of the C API), Python, Java, JavaScript (through WebCL), D, and .Net. The host code is used to setup the devices, load and compile the kernels, control the movement of data, and execute the kernels. The OpenCL host API is relatively low level and the programmer must specify all device selection, setup, allocation of resources and so forth. The OpenCL kernel code is the code that is executed in a parallel manner on the OpenCL device. The kernel can generally be thought of as the inner most loop of sequential code, something that needs to be run numerous times independently of any other iteration. It can be written in a data parallel style (best for GPU's) or task parallel style and is written in a language provided by the OpenCL specifications. The main strength of OpenCL is its portability across accelerators and vendor's platforms. OpenCL programs can run on different devices: GPUs, CPUs, Cell broadband engines, and other accelerators. OpenCL is supported by multiple vendors, i.e., Intel, AMD, IBM, and NVIDIA. For the portability sake, OpenCL uses a system of dynamic, run-time compiling of the kernel source code and compliers built into the API. This gives a reasonable guarantee that as the compilers improve and the devices change, the OpenCL kernels will still perform well. The drawback, at least for the commercial community, is that without other tools the source code is available for easy inspection.

5. METHODOLOGY

5.1 OpenCL implementation

Only the C-based API specification for controlling the computation and the C99 standard based kernel language for the compute intensive calculations are considered in this work. Within this specification, we developed our OpenCL program for matrix multiplication module primes by designing, implementing, and integrating three main software components as shown in Figure 1.

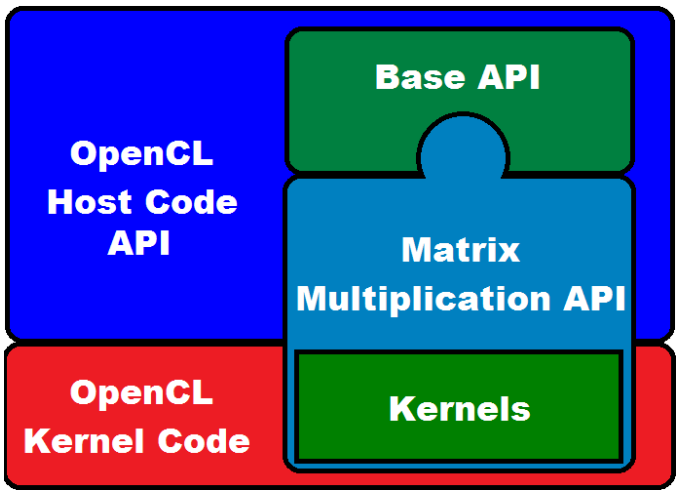


Figure 1. OpenCL framework.

The generic base API gathers and handles all the OpenCL setup and basic functions. It was designed and implemented so that it can be extended for different purposes. It is in charge of: (1) enumerating all the OpenCL platforms available; (2) selecting the most powerful device based on a weighted score; (3) creating a compute context and command queue; and (4) keeping track of all memory buffers. The API contains basic functions for the manipulation of memory buffers and

the retrieval of basic information about the selected device. Table 1 summarizes the main functions we implemented for controlling the OpenCL device.

Table 1. Main base API functions controlling the API device.

Buffer manipulation functions	Description
create_buffer()	Create an empty buffer on the OpenCL device with a specified size
load_buffer()	Load a specified buffer from a host pointer that has a specified number of bytes
create_and_load_buffer()	Create a buffer with a specified size and load it from a host pointer
read_buffer()	Read a specified number of elements back to a host pointer from a specified buffer
delete_buffer()	Deallocate a specified buffer
copy_buffer()	Copy the contents of one buffer to another buffer
Status functions	**Description**
get_mem_capacity()	Return the total amount of memory available on the OpenCL device
get_mem_free()	Return the amount of memory not currently allocated
get_buffer_size()	Return the size of a specified buffer
get_errcode()	Return the last error code the API recieved from the OpenCL API

The matrix multiplication module in Figure 1 extends the base API functionality and contains matrix multiplication specific functionality. It also handles all loading and compiling of kernels. Table 2 summarizes the functions we implemented to allow the computation of exact matrix multiplication on the OpenCL device.

Table 2. Main base API functions controlling the API device.

Buffer manipulation functions	Description
create_and_load_matrix_buffer()	Create a buffer and load it from a host pointer, padding the contents to the appropriate format
load_matrix_buffer()	Load a buffer from a host pointer, padding the contents to the appropriate format
read_matrix_buffer()	Read a buffer back to a host pointer, depadding the contents
Kernel call functions	**Description**
add_matrix()	Add two matrix buffers together, placing the results in a third buffer
sub_matrix()	Subtract two matrix buffers from each other, placing the results in a third buffer
mod_mul_matrix()	Multiply two matrix buffers together using the *basic modulus algorithm*, placing the results in a third buffer
delay_mod_mul_matrix()	Multiply two matrix buffers together using the *delayed remainder modulus* , placing the results in a third buffer
partial_8_mul_matrix()	Multiply two matrix buffers together using the *partially delayed remainder* with delay 8, placing the results in a third buffer
partial_16_mul_matrix()	Multiply two matrix buffers together using the *partially delayed remainder* with delay 16, placing the results in a third buffer
partial_32_mul_matrix()	Multiply two matrix buffers together using the *partially delayed remainder* with delay 32, placing the results in a third buffer
partial_1024_mul_matrix()	Multiply two matrix buffers together using the *partially delayed remainder* with delay 1024, placing the results in a third buffer

The kernel code implements basic $O(n^3)$ matrix multiplication with a modulus operation every 1, 8, 16, 32, or 1024 summands. Each work-item works on one element of the result matrix and each work-group works on a 16x16 block of the result matrix. The kernel was optimized for NVIDIA GPU architecture; it uses local memory to increase throughput; it requires padding of the matrix (with zeroes) to have dimensions that are a multiples of 16 to eliminate the need for expensive branching (handled by API); and its work-group size was tuned for this specific architecture. There are more

efficient algorithms for non-modular multiplication for GPUs[15] that could have been modified but we chose to ignore those approaches for now because of portability concerns. For CPU devices in OpenCL the code is auto-vectorized to take advantage of x86 SSE instructions.

5.2 CUDA implementation

The CUDA implementation used in this paper was not as extensive as the OpenCL implementation. This was possible because CUDA handles a lot of the low level setup and resource allocation in the background for the programmer. The host code is a straightforward single purpose program with one version of each of the three matrix multiplication modulo prime algorithms considered. The kernels themselves are pure translations of the OpenCL kernels so that the implementation would not be biased towards non-IEEE compliant optimizations that CUDA allows, unless the compliers are given different switches. Shared memory is used to cover main memory latency. Each thread works on a single element of the result and each thread-block works on a 16x16 block of the result; the input is also padded to dimensions that are multiples of 16 to avoid branching.

6. RESULTS

6.1 Test setup

Our comprehensive analysis of performance vs. portability and of performance vs. accuracy relies on this testing environment. We ran our tests on two different NVIDIA GPUs, the consumer GPU card GeForce GTX 480 and the computing GPU card Tesla C2050, running the 285.05.33 drivers. The host computer was a high-end workstation with dual quad-core Intel Xeon E5520 processors. The host Operating System was CentOS 5.7.

We repeated each test ten times and reported in this paper the average values over the ten runs. In all cases we did not observe large deviations and therefore we do not report the standard deviation.

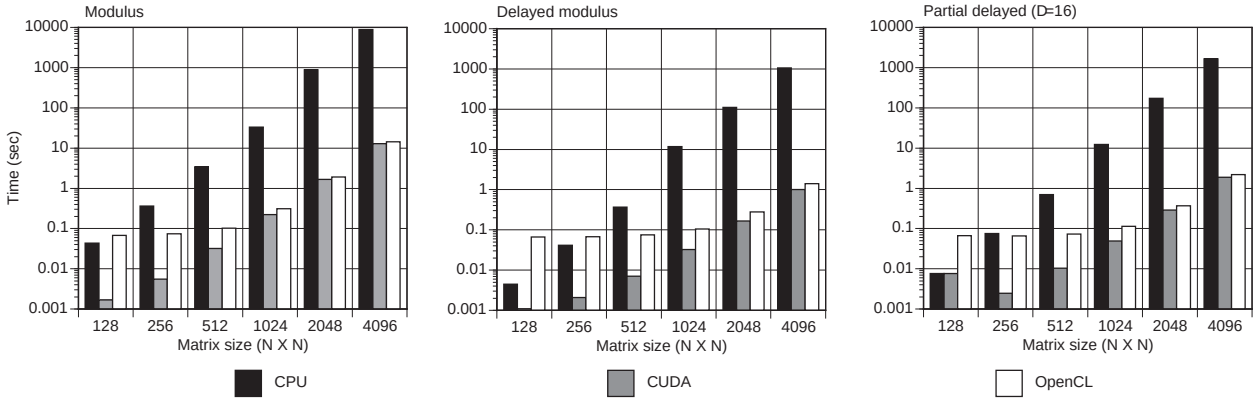
6.2 Performance versus portability

To quantify the trade-off performance versus portability associated with different types of programming languages supporting GPUs, we compared the performance of the matrix multiplication modulo primes for three different algorithms, i.e., the modulus, the delayed modulus, and the partial delayed modulus algorithms. We integrated each algorithm in C code for CPU, a CUDA code for NVIDIA GPUs, and a OpenCL code for GPUs and multi-core processors. The CPU version was unoptimized and naive and thus should not be considered the best performance that can be expected from pure CPU implementations. It was mostly used to confirm that the CUDA and OpenCL versions were correct and is reported to give an idea of performance that can be expected with the same development time to the CUDA and OpenCL versions.

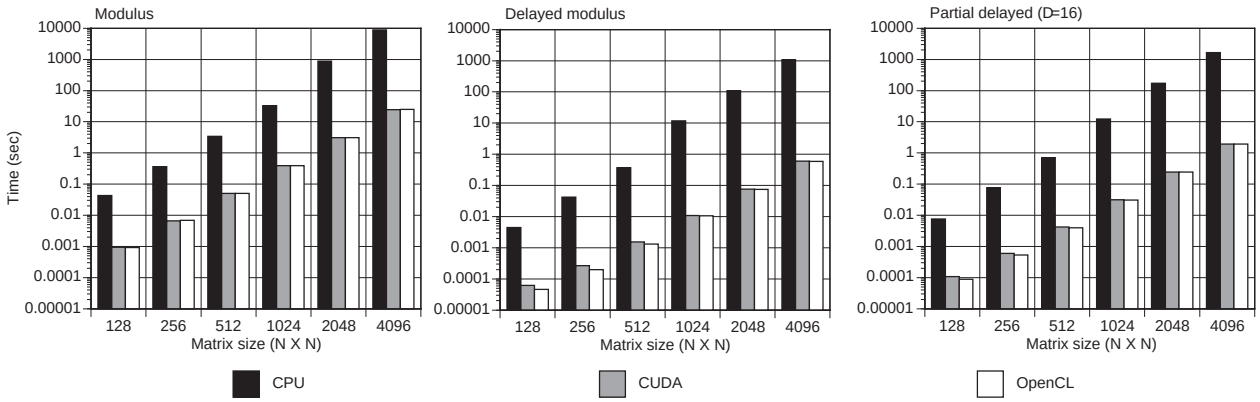
Initial observations of test performance on a GTX 280 graphics card and an old driver, i.e., 270.41.19, indicated some difference in terms of performance between the CUDA and OpenCL implementations. The CUDA execution times for the three algorithms were consistently faster, although the difference was not significant for large matrices (see Figure 2.a). Recently improvements in the CUDA drivers has resulted in a substantial improvements of the OpenCL performance for the three algorithms. Tests with 285.05.33 drivers indicate no substantial differences between the CUDA and OpenCL executions for large matrices. Figure 2.b and Figure 2.c shows the execution times of the three algorithms on a GTX 480 and an Tesla C2050. At the same time the three figures outline the substantial gain in performance when executing the calculations on GPUs rather than on CPUs.

6.3 Performance versus accuracy

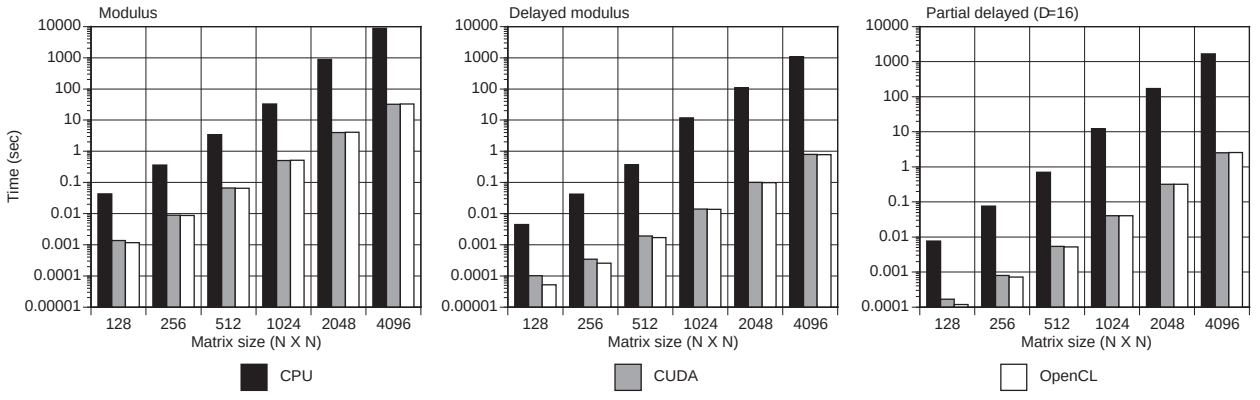
To quantify the trade-off between performance and accuracy associated with the use of double rather than single precision, we compared the performance of the matrix multiplication modulo primes p using single or double precision and for five different algorithms: the modulus, the delayed modulus, the partial delayed ($D = 8$) modulus, the partial delayed ($D = 16$) modulus, and the partial delayed ($D = 32$) modulus. Note that the algorithms are identical with double precision except that a larger range of primes (p is in the range of $2..2^{26}$) can be represented. The *delayed remainder modulus* algorithm in this case requires $wA * p^2 < 2^{53}$ and the *partially delayed remainder* algorithm requires $2^i * p^2 < 2^{53}$ to remain within the range exactly representable. As for the previous tests, we considered the two GPUs, i.e., the GTX 480 (Figure 2.a) and the C2050 (Figure 2.b), however we ran the tests only with OpenCL because of its portability across other platforms than the ones considered in this paper. We considered matrices with size ranging between 16 X 16 elements and 8192 X 8192



(a) GTX 280



(b) GTX 480

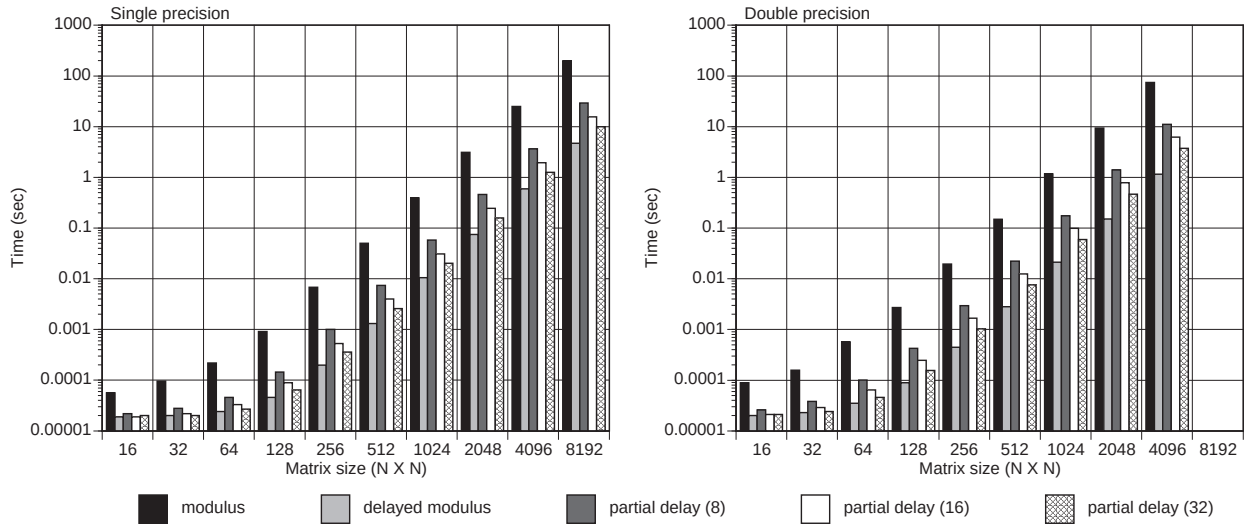


(c) C 2050

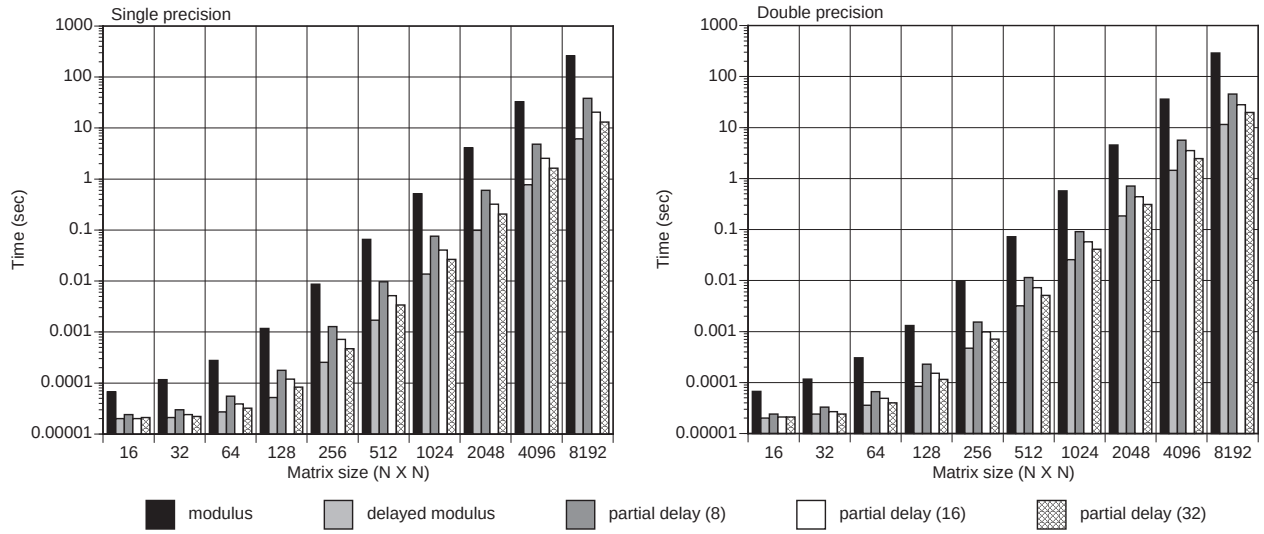
Figure 2. Performance vs. portability trade-off is measured by comparing the execution time of three matrix multiplication modulo primes on three different GPU cards by using a C code for CPU, CUDA on GPUs, and OpenCL o9n GPUs.

elements. Note that for the double precision tests on the GTX 480 we could not run the test with the largest matrix because of memory constraints: this GPU has only 1.5GB memory versus the 2.62 GB in the C2050 and the maximum size of a buffer is limited to a quarter of the available memory.

As expected, the execution times present a cubic growth. The modulus algorithm is the more time demanding in all four cases (i.e., with and without double precision on both GPUs) while the delayed modulus is the less time demanding.



(a) GTX 480 (Fermi chip)



(b) C2050 (Fermi chip)

Figure 3. Performance vs. accuracy trade-off is measured by comparing the execution time of five matrix multiplication modulo primes on two different GPU cards by using single and double precision.

The three partial delay algorithms have times decreasing with the frequency of the partial delay. The time for the partial delayed with $D = 8$ is larger than the time for the partial delayed with $D = 16$ and the latter time is larger than the time for the partial delayed with $D = 32$.

We observed that the use of single or double precisions has a more tangible impact in execution times for consumer cards than for compute cards: the execution times for the GTX 480 with double precision are much higher than the times on the same card with single precision (Figure 3.a). This is not the case for the same calculations on the Tesla C2050 (Figure 3.b). In general, when considering compute intensive calculations, for consumer cards such as the GTX 480, the single precision / double precision performance ratio is nominally 8:1 due to hardware limitations. For the Tesla cards (e.g., the C2050) the ratio is 2:1. These ratios decrease as the calculations become I/O bound, which can easily happen when the computation time between memory accesses does not cover the high access latency. In our tests on the GX 480, we observed a ratio of 3:1 for the modulus algorithm, a ratio of 2:1 for the delayed modulus algorithm, and a ratio of approximately 3:1 for the three partial delayed algorithms. When repeating the same tests on the Tesla card, we observed

smaller ratios, i.e., 1.1:1 for the modulus algorithm, a ratio of 1.8:1 for the delayed modulus algorithm, and a ratio ranging from 1.1:1 to 1.5:1 for the three partial delayed algorithms. This has very positive implication that for compute cards the performance/accuracy trade-off is negligible. This most likely arises from the calculations becomming I/O bound and the slower double precision operations hiding the access latency that would become idle time with single precision.

7. CONCLUSION

We have shown that with current implementations there is minimal trade-off with respect to performance/portability with current NVIDIA drivers and hardware when choosing OpenCL over CUDA. All of the performance is still available with the added benefit of cross-vendor portability. This was not the case with earlier drivers.

With respect to performance/accuracy, we have shown that with compute oriented GPUs there is only a small trade-off when the added accuracy of double precision is necessary and the calculations become I/O bound. There is a higher cost for consumer cards but we still believe the cost is acceptable and it is of course necessary when larger modulo require double precision.

8. ACKNOWLEDGMENTS

Research supported by National Science Foundation Grant CCF-0830130 and Air Force Small Business Technology Transfer (STTR) funding.

REFERENCES

[1] Lenstra, A. K., Lenstra, H. W., Manasse, M. S., and Pollard, J. M., "The number field sieve," in [*Proc. of 22nd ACM Symp. Theory Comput.*], 564–572 (1990).

[2] Montgomery, P., "A block Lanczos algorithm for finding dependencies over GF(2)," in [*Proc. of Advances in Cryptology-Eurocrypt*], *Lecture Notes in Computer Science* **921**, 106–120 (1995).

[3] May, J., Saunders, B., and Wan, Z., "Efficient matrix rank computation with application to the study of strongly regular graphs," in [*Proc. of 2007 Internat. Symp. Symbolic Algebraic Comput. (ISSAC07)*], 277–284 (2007).

[4] Weng, G., Qiu, W., Wang, Z., and Xiang, Q., "Pseudo-Paley graphs and skew Hadamard difference sets from commutative semifields," (2006). Preprint.

[5] Linusson, S., J.Shareshian, and V.Welker, "Complexes of graphs with bounded matchings size," preprint (2004).

[6] Björner, A. and Welker, V., "Complexes of directed graphs," *SIAM Journal on Discrete Mathematics* **12**(4), 413–424 (1999).

[7] Bjöerner, A., Lovász, L., Vrećica, S., and Živaljević, R. T., "Chessboard complexes and matching complexes," *Journal of the London Mathematical Society, II* **49**(1), 25–39 (1994).

[8] Bennett, J. C., Day, D. M., and Mitchell, S. A., "Summary of the csri workshop on combinatorial algebraic topology (cat): Software, applications, and algorithms," *SANDIA REPORT SAND2009-7777* (2010).

[9] Adams, J., Saunders, B. D., and Wan, Z., "Signature of symmetric rational matrices and the unitary dual of Lie groups," in [*Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC05)*], (2005).

[10] Stembridge, J. R., "Explicit matrices for irreducible representations of Weyl groups," *Represent. Theory (electronic)* **8**, 267–289 (2004).

[11] May, J., Saunders, B. D., and Wood, D. H., "Numerical techniques for computing the interta of products of matrices of rational numbers," in [*Proc. of the International Symbolic-Numeric Computation Workshop (SNC07)*], 125–132 (2007).

[12] Stanford Computer Graphics Labratory, "BrookGPU." `http://graphics.stanford.edu/projects/brookgpu/index.html`.

[13] *CUDA API Reference Manual: Version 4.1* (2012).

[14] *The OpenCL Specification: Version 1.2* (2011).

[15] Nath, R., Tomov, S., and Dongarra, J., "An Improved MAGMA GEMM for Fermi GPUs," *University of Tennesse Computer Science Technical Report, UT-CS-10-655 (also LAPACK working note 227)* (July 2010).