Experiments with Massively Parallel Matrix Multiplication

Timothy W. O'Neil
Dept. of Computer Science, The University of Akron, Akron OH 44325 USA
toneil@uakron.edu

Abstract

This paper presents initial experiments in implementing two notable matrix multiplication algorithms – the DNS algorithm and Cannon's algorithm – using NVIDIA's general-purpose graphics processing units (GPGPUs) and CUDA development platform. We demonstrate that these implementations are comparable with traditional methods in terms of computational expense and may scale better than traditional techniques.

1 Introduction

Many important problems in science and engineering rely on the underlying data structure of the *matrix*, a two-dimensional array of numbers. So important is the efficient manipulation of these data structures that a great deal of effort has gone into devising clever algorithms for efficiently multiplying matrices using either mathematical tricks or special properties of the underlying computing platform. Two such methods are:

- the Dekel-Nassimi-Sahni (DNS) algorithm [1] for multiplying matrices on a hypercube, and
- Cannon's algorithm [2] for multiplying matrices on a torus (i.e. a mesh with wraparound links).

Currently, the use of general-purpose graphics processing units (GPGPUs) is becoming increasingly important to the high-performance computing field as an inexpensive way to apply large numbers of processing cores to efficiently solving complex computational problems. Despite this trend, the traditional matrix multiplication algorithm remains the method of choice, with no consideration heretofore of alternatives that may be better suited to the GPGPU model. In this paper we present an initial exploration of such techniques in this promising new programming paradigm.

In the next section we review details of the algorithms and of the GPGPU programming model. We then detail our experiments and results. Finally, we point to future extensions of this work.

2 Background

The mathematical concept of a matrix is a regular topic in undergraduate classes. The algorithms and details of the CUDA platform are also widely discussed in the literature. We briefly review these concepts now for completeness' sake.

2.1. Matrix Multiplication

Assume that **A** and **B** are *n*-by-*n* square matrices. The traditional algorithm for multiplying matrix **A** times matrix **B** to derive matrix **C** is to compute all dot-products of row vectors from **A** and column vectors from **B**. Classically this is represented by the formula

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j].$$

Based on this formula the classic method for performing this computation is then to use three nested loops to step through the rows of **A** and the columns of **B**, multiplying individual components and summing the products to produce an individual element of **C**:

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

The computational expense of this $\mathbf{O}(n^3)$ algorithm has motivated many researchers to explore the use of multiple processors/threads to reduce the cost of matrix multiplication.

2.2 The DNS algorithm

One such alternative is the DNS algorithm. We review its details based on the treatment of the subject

978-1-880843-95-6 / copyright ISCA, CATA 2014
March 24-26, 2014, Las Vegas, Nevada, USA

found in [3]. The algorithm utilizes n^3 processors organized in a 3-dimensional n-by-n-by-n grid as depicted in Figure 1 to hold all terms of all inner products simultaneously. On each plane k ($0 \le k < n$), processor (i, j) ($0 \le i, j < n$) holds the terms **A**[i][k] and **B**[k][j]. (In other words plane k contains the k^{th} row of **A** and the k^{th} column of **B**.) After multiplying them together, an all-to-one reduction along the k axis takes place so that process (i,j) in plane zero accumulates all terms and computes **C**[i][j]. Thus the desired product matrix **C** can ultimately be derived by collecting its individual elements from the processors on plane zero of the grid.

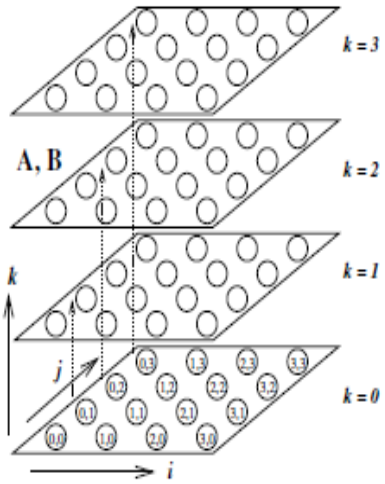

Figure 1: The 3-D grid used by DNS to multiply 4-by-4 matrices.

The DNS algorithm was originally designed for processors connected via hypercube, allowing for efficient distribution and collection of data. In the beginning elements **A**[i][j] and **B**[i][j] are contained in processor (i, j) on plane zero. Because of the independent communication channels, row k of **A** is transmitted in constant time to the corresponding processors in row k on plane k. Simultaneously column k of **B** is sent to the processors in column k on plane k. Once in place, these values can then be broadcast in logarithmic time [3] to the other processors in plane k until each holds their expected matrix elements. Following multiplication, the reduction/gather from plane k ($k > 0$) to plane zero can also be completed in logarithmic time [3]. Thus the DNS algorithm can claim fast computation time, but at great cost.

2.3 Cannon's algorithm

As a contrast to our DNS experiments and to the traditional three-nested-loops algorithm we also consider a GPU implementation of Cannon's algorithm. The details which follow are derived largely from the treatment in [4].

Cannon's algorithm replaces the traditional sum seen above with

$$C[i][j] = \sum_{k=0}^{n-1} A[i][(i+j+k)\ \%\ n] \cdot B[(i+j+k)\ \%\ n][j]$$

(where **%** is the modulus operator) to create a memory efficient algorithm. Each processing core in an n-by-n mesh holds one element of **C**, as seen in Figure 2. Each also holds individual elements of **A** and **B**. The specific starting elements held are a function of the processor's place in the grid. After these are multiplied and added into the partial sum the **A** element is shifted one place to the left and the **B** element shifted one place upward, as depicted in Figure 2. Simultaneously new elements of **A** and **B** are received from neighbors to the right and below, respectively, multiplied and added into the partial sum. After n such shifts the final matrix element is obtained.

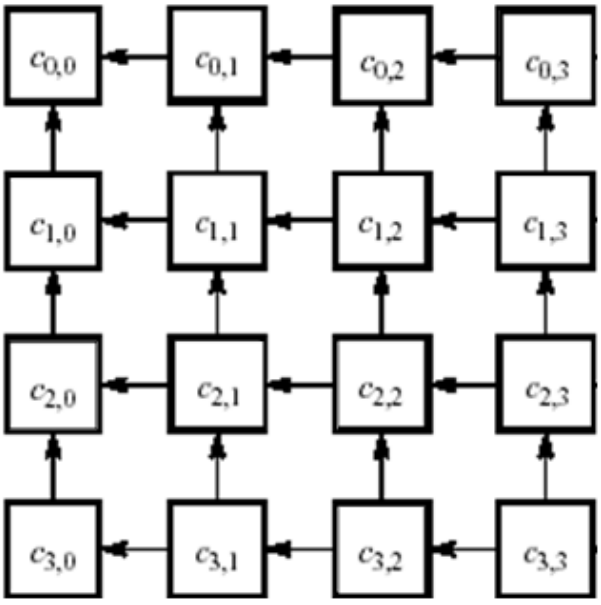

Figure 2: Mesh used by Cannon's algorithm to multiply 4-by-4 matrices.

2.4 The CUDA Platform

The Compute Unified Device Architecture (CUDA) is the programming environment developed by NVIDIA which permits programming of general-purpose graphics processing units (GPGPUs) directly in a high-level language such as C. A typical architecture for an early generation CUDA-capable GPU appears in Figure 3 below [5]. As can be seen, the processor consists of

some number of symmetric multiprocessors (SMPs), each with 8 cores. (Later generation cards, including the Tesla card used in our experiments, have 14-15 SMPs each with 32 cores.) Each SMP contains a common global memory shared by the cores, as well as registers, texture memory and shared memory. (The Tesla card used in our experiments contains 49 kilobytes of shared memory per SMP and 1 gigabyte of global memory.)

.

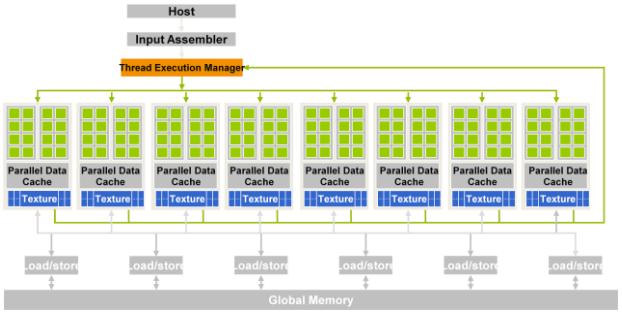

Figure 3: A CUDA-capable GPU architecture

A CUDA programmer views a program's execution as consisting of a *warp* of threads running in parallel on an SMP. Such threads are visualized as in Figure 4 [5]. A CUDA program creates a *grid* consisting of multiple *blocks* of threads. (The design of the Tesla card used in our experiments permits up to 1024 threads per block.) Each thread executes code in the *kernel* using data from the global device memory. Since each grid, block and thread is uniquely addressable within a CUDA program (as shown), each thread executes the same kernel on different data sets, leaving the user with the view of a massively parallel SIMD processor.

For example, the authors in [5] implement the classic matrix multiplication method expressed above by dividing the matrices **A**, **B** and **C** into tiles of width *w*. They then launch an $\frac{n}{w} \times \frac{n}{w}$ grid of $w \times w$ blocks, each of which computes one tile of **C**. The individual thread identifiers are used to derive the correct *i* and *j* values, leaving only the *k* loop to be stepped through.

3 Experimental Models

Given the large number of processing cores, the CUDA platform would appear to be the ideal venue for the DNS algorithm. Almost all CUDA programs follow the same basic pattern: transfer data to global memory, start the threads executing the kernel, then transfer the results from the CUDA card's global memory back to RAM. The key to such experiments then becomes designing a good kernel and properly organizing the

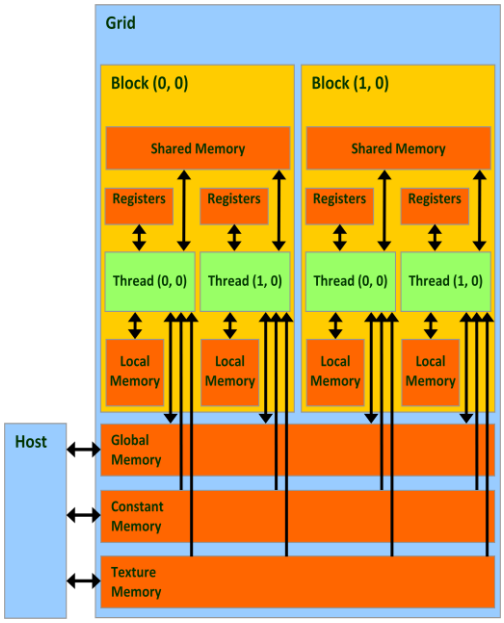

Figure 4: CUDA Thread and Memory Organization

thread blocks and grids. Our experiments fall into two classes.

3.1. Single Thread Block

Initially we chose to experiment with a single thread block comprising the entire 3-D DNS grid. Because of the upper limit on threads per block, this necessarily limited the size of the *n*-by-*n* matrices we could experiment with to those with $n \leq \sqrt[3]{1024} \approx 10.08$. We thus use randomly generated 8-by-8 matrices for this portion of the testing.

When started, each thread in the block is assigned a 3-dimensional identifier (*tx*, *ty*, *tz*). The kernel code looks like this:

```
if (tz == 0) {
  sA[tx][ty] = A[tx][ty];
  sB[tx][ty] = B[tx][ty];
}
__syncthreads();

term[tx][ty][tz] = sA[tx][ty] * sB[ty][tz];
__syncthreads();

if (ty <  4) {
  term[tx][ty][tz] += term[tx][ty + 4][tz];
  if (ty < 2)
    term[tx][ty][tz] += term[tx][ty + 2][tz];
  if (ty == 0)
    C[tx][tz]
     = term[tx][ty][tz] + term[tx][ty + 1][tz];
}
```

In this implementation the *y*-dimension of the thread block is analogous to the *k*-dimension of the original version of Figure 1, while *i* and *j* there match with *x* and *z* here, respectively. To begin the threads collaboratively load the matrices **A** and **B** from global memory into shared memory so that future accesses will be faster. After each thread has computed its individual term, the reduction is performed. As noted in [6], since the final reductions involve 32 or fewer threads on each rectangular plane, there is no need for explicit synchronizations, which should speed execution. Furthermore, as suggested there, execution time is reduced by listing each reduction separately rather than using a small loop.

Similarly our implementation of Cannon's algorithm utilizes one 8-by-8 grid, with thread (*tx*, *ty*) executing the kernel code below. After collaboratively loading **A** and **B** into shared memory, thread (*tx*, *ty*) steps through the elements of row *ty* and column *tx* in the order prescribed by [2], multiplying the pairs and accumulating the products into a local register. When finished it writes its result back to **C** in global memory in the proper location.

```
float localVal = 0;
sA[ty][tx] = A[ty][tx];
sB[ty][tx] = B[ty][tx];
__syncthreads();

for (int k = 0; k < n; ++k) {
   int dex = (tx + ty + k) % n;
   localVal += sA[ty][dex] * sB[dex][tx];
}
__syncthreads();
C[ty][tx] = localVal;
```

While these implementations are based on the classic ones, we can expect the Cannon's implementation to perform better than the 1-block DNS version due to fewer needed warp launches and less thread synchronization. We can also see that, while more threads are utilized in DNS and therefore more effective use of the CUDA card is made, each thread does less work (e.g. one multiply versus eight). A reimagining of the base algorithms is in order as part of future extensions.

3.2.Multiple Thread Blocks

The advantage of having all threads in one block in a CUDA program is that those threads can synchronize with each other. Threads in different blocks cannot [5]. On the other hand, the existence of a maximum thread count of a single block compels us to experiment with multi-block implementations of DNS.

The first of these separates each plane seen in Figure 1 into its own block, creating an *n*-by-1 grid of *n*-by-*n* blocks. Each block computes the column of **C** corresponding to its block number. As before, this requires collaboratively reading the entire **A** matrix into shared memory. However, we now only require a single column of **B** to complete a block's calculations. Otherwise this version of DNS proceeds as described above, with each thread computing its individual element, followed by a reduction into one summation that is written to the proper location in global memory.

We also logically extend this idea and study a DNS implementation with a 2*n*-by-1 grid of ½ *n*-by-*n* blocks, and one with a 4*n*-by-1 grid of ¼ *n*-by-*n* blocks. The first version computes 4 columns of the product matrix per block, the second, 2. Note that this type of arrangement makes sense because of the core arrangement inherent in the DNS algorithm and would not be appropriate for Cannon's algorithm.

4 8x8 Experimental Results

Our results are summarized in Table 1 below. All experiments were conducted using a Dell server containing a Tesla C2070 CUDA card, with times given in microseconds. As stated above, this card contains 14 SMPs with 49 kilobytes of shared memory per MP. All SMPs share 1 gigabyte of global memory.

Table 1. Experimental results, 8-by-8 matrix multiplication.

Algorithm	Trials	Min	Max	Avg	StD
Cannon	15	277.5	287.7	281.6	3.2
DNS, 8x8x8 block	24	279.7	291.2	282.7	3.1
DNS, 8 8x8 blocks	26	277.0	287.0	282.2	2.6
DNS, 16 4x8 blocks	23	278.7	290.1	282.3	2.6
DNS, 32 2x8 blocks	25	280.2	291.5	283.1	2.7

We see that the preferred methods appear to be Cannon's algorithm (as expected) and the DNS implementations with fewer blocks, with added warp launches and thread synchronization adding an extra 0.5 – 1.5 microseconds to execution times. Because of the tiny size of the experimental matrices this advantage is very small, with speedups for Cannon over the DNS implementations ranging from 0.2% to 0.5%. That said, the range of numbers and their stability (as represented by the small standard deviation) indicate that the 8- or 16-block DNS is a better choice than DNS with other

grid/block configurations as we move on to larger experiments.

5 Increasing Matrix Sizes

As indicated above, the shared memory of the Tesla card is just large enough to contain 3 64-by-64 integer matrices. Even with the efficient use of memory in our Cannon's algorithm implementation, we could only hold 2 110-by-110 integer matrices in shared memory. We thus are unable to use our algorithms as expressed on large matrices and must consider the question of how to extend our work to handle them.

As indicated above the answer is to tile our matrices, as shown in Figure 5 below. As seen there, multiplying 16-by-16 matrices would require computing 8 8-by-8 sub-products and accumulating the results in global memory. It is straight-forward to increase the number of blocks by a factor of 8 and assign each to one of the sub-products based on its block i.d. This technique will work up to the CUDA-imposed maximum of 65,535 blocks per grid. The inherent scalability of CUDA programs and the large number of resources available to us on a CUDA card makes this very feasible.

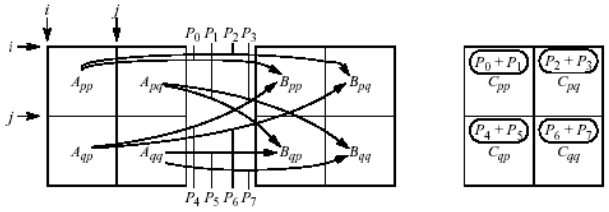

Figure 5. 2-by-2 block matrix multiplication [4].

The chief implementation complications involve writes to global memory. With the 8-by-8 trials we simply overwrote the contents of memory locations. In the larger trials, we have to:

- explicitly set C to zero in global memory initially, and
- use atomic adds to accumulate the final product, since there is not explicit synchronization among the threads in different blocks writing back to the same location.

The means for doing both of these are built into the CUDA API [5] and are easy modifications to existing code once discovered.

We briefly examine our kernel code for Cannon's algorithm to illustrate. Partition the 16-by-16 product matrix into 8-by-8 sub-products. (In the code below n = 16 and *TILE_WIDTH* = 8.) Next create a 4-by-2 grid, with 2 adjacent blocks on a row collaborating on one sub-product. Each thread now computes its place in the larger matrices and proceeds as before, computing the dot-products for its assigned vectors. At the end, atomic operations are used to accumulate the results in global memory.

```
// for block (bx, by), thread (tx, ty)
int divisor = n / TILE_WIDTH;
int offset = bx % divisor;
bx = bx / divisor;
int Row = by * TILE_WIDTH + ty;
int Col = bx * TILE_WIDTH + tx;

float Pvalue = 0;
sA[ty][tx]
   = A[Row * n + (offset * TILE_WIDTH + tx)];
sB[ty][tx]
   = B[(offset * TILE_WIDTH + ty) * n + Col];
__syncthreads();

for (int k = 0; k < TILE_WIDTH; ++k) {
   int dex = (tx + ty + k) % TILE_WIDTH;
   localVal += sA[ty][dex] * sB[dex][tx];
}
__syncthreads();

atomicAdd(&C[Row * Width + Col],localVal);
```

6 16x16 Experimental Results

The results of our initial experiments are summarized in Table 2 below. They were conducted with the same equipment used for the 8-by-8 trials. Having concluded previously that implementations with fewer thread blocks would be more efficient, we focused on only our first three proposed designs.

Table 2. Experimental results, 16-by-16 matrix multiplication.

Algorithm	Trials	Min	Max	Avg	StD
Cannon	15	270.4	277.5	274.1	2.0
DNS, 8x8x8 block	15	273.4	283.8	277.4	3.4
DNS, 8 8x8 blocks	16	282.7	300.8	288.2	4.6

The results indicate that not only does octupling the amount of work not dramatically add to the overall computation time, the scalability of the algorithms is so good that there is marked improvement. While octupling the number of active blocks and threads, the 16-by-16 Cannon demonstrated a 2.7% speedup over its 8-by-8 counterpart. Similarly 16-by-16 uniblock DNS is 1.9% faster than its 8-by-8 equivalent. The 8-block DNS algorithm is 2.1% slower, likely due to the large number

of thread blocks (64) interacting to solve such a small problem.

We also see that the advantage of Cannon's algorithm over DNS is now more noticeable, with a speedup of 1.2%. There are numerous confounding factors in such an experiment, such as poor implementation and trials conducted on equipment at different times of the day or year. While a great deal more work remains to be done with larger matrices before a definitive answer can be found, we are forced to conclude that Cannon's algorithm is the preferred method for matrix multiplication using CUDA over DNS.

7 Conclusion

We began this investigation with a search for alternatives to the traditional nested-loops matrix multiplication method of [5] that take better advantage of the massive amount of resources held by a GPGPU. Two established options became apparent, Cannon's algorithm and the DNS method. In this paper we have demonstrated that, for small examples, Cannon's algorithm as implemented is faster. While the DNS algorithm makes more efficient use of more resources, the complex synchronization among that many threads adds too much overhead.

We have also discovered that there is a balancing act taking place. Too many threads in fewer blocks slow an implementation because of synchronization costs within a thread block. On the other hand, too few threads in many blocks also slow the program down because of details hidden in CUDA's warp scheduling algorithm. Striking the right balance is key to developing effective CUDA code. We have observed some trends as part of this investigation, but a systematic way of deciding the correct division of blocks and threads for maximum effectiveness must be developed.

While the pattern is established, more trials with larger matrices are needed for verification. Additionally, developing potential alterations to the basic details of the explored algorithms for best execution in the CUDA environment remains an open problem.

Acknowledgements

Financial support for this work provided by the University of Akron's Buchtel College of Arts and Sciences. Additional support provided by the NVIDIA Corporation through their CUDA Teaching Center program. The author also thanks Charles Van Tilburg for his help with this work in his capacity as technical support professional for the UA Computer Science department.

Finally, the author thanks the students in the Fall 2013 section of Introduction to Parallel Processing for their help in generating experimental results, particularly Darrick Ferrell and Evan Purkhiser. Others from the class who contributed include Alexander Addy, Jonathan Bletso, Andrew Edwards, Michael Griffith, Michael Gruesen, Rob Herman, J. D. Kilgallin, Brad Santoro, Steve Shumway, Robert Smith, Andrew Tilisky, Sam Trela and Walter Wolfe.

References

[1] Dekel, E., Nassimi, D. and Sahni, S. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, Vol. 10, pp. 657 - 673. 1981.

[2] Cannon, L.E. *A Cellular Computer to Implement the Kalman Filter Algorithm*. Ph.D. thesis, Montana State University, 1969.

[3] Grama, A., Gupta, A., Karypis, G. and Kumar, V. *Introduction to Parallel Computing*. Harlow England: Pearson Education Ltd., 2003.

[4] Wilkinson, B. and Allen, M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, N.J.: Pearson Education, Inc., 2005.

[5] Kirk, D.B. and Hwu, W.-M. W. *Programming Massively Parallel Processors: A Hands-On Approach*. Waltham MA: Morgan-Kaufmann/Elsevier Inc., 2013.

[6] Harris, M. Optimizing Parallel Reduction in CUDA. *http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf*. Online, accessed 22 October 2013.