Exploring Applications in CUDA

Michael Kubacki

Computer Science and Engineering, University of South Florida

mkubacki@mail.usf.edu

***Abstract*—Modern Graphics Processing Units (GPUs) are capable of much more than supporting GUIs and generating 3D graphics. These devices are highly parallel, highly multithreaded multiprocessors harnessing a large amount of floating-point processing power for non-graphics problems. This project is based on experiments in CUDA C. These examples seek to demonstrate the potential speedups offered by CUDA and the ease of which a new programmer can take advantage of such performance gains.**

***Keywords*— CUDA, GPGPU, Parallel Programming, GPU**

I. Introduction

Traditionally, software applications have been written in a sequential fashion, easily understood by a programmer stepping through the code. Software developers have largely relied on increasing clock frequencies and advances in hardware to simply speed up execution of these programs. However, it is no longer feasible to expect a single processor core to provide ample speedup over previous generations. Physical constraints such as energy-consumption and heat-dissipation, have forced CPU manufacturers to fabricate multi-core processors incorporating an increasing number of processing cores. Programming such devices bears new challenges and opportunities for the software industry, generating a strong desire for parallel applications to take advantage of the new cores available.

The ubiquitous graphics processing unit (GPU) has evolved from hardwired, limited capability VGA controllers to a highly parallel, highly multithreaded multiprocessor. The capabilities of modern GPUs lie far outside the historic graphics domain. To contrast the parallelization offered among modern multiprocessors and GPUs, consider the Intel Core i7 multiprocessor. It has four processor cores and supports hyperthreading with two hardware threads. The NVIDIA GeForce GTX 285 GPU boasts 240 cores, each a heavily multi-threaded processor [6]. The multi-core paradigm strives to maintain execution speed of sequential programs and move into multiple cores. While many-core devices, such as modern GPUs, seek to increase execution throughput of parallel applications [4].

As the shift to parallel computing takes place, GPU computing is particularly attractive because of the massive parallelization, floating-point processing power, and market presence offered by GPUs. However, not all applications require such capabilities. Optimal performance can be achieved by carefully examining which sections of an application are suitable for sequential execution on the CPU or massive parallelization on the GPU. Of course, after such analysis, an efficient implementation is necessary. This paper explores two applications based on the CUDA programming model and a sequential application demonstrating the necessary preparation for a parallel implementation.

II. Background

A. NVIDIA CUDA Programming Model

NVIDIA's CUDA (Compute Unified Device Architecture) is a scalable parallel programming model and software platform for the GPU and other parallel processors that allow the programmer to bypass the graphics API and graphics interfaces of the GPU and simply program in C or C++. CUDA uses a SPMD (single-program, multiple data) style, programs are written for one thread that is instanced and executed by many threads in parallel on the multiple processors of the GPU [7]. Essentially, CUDA C is an extension to the C/C++ languages and can be understood by a knowledgeable programmer and as already mentioned, no knowledge of graphics programming is needed.

CUDA made its debut in November, 2006 on the NVIDIA GeForce GTX 8800 and every NVIDIA GPU since has been built CUDA-enabled [9]. The features supported by a given NVIDIA CUDA-capable GPU are called its compute capability. The example programs presented in this paper require a compute capability of at least 1.2.

Throughout this paper, the CPU will be referred to as the host and the GPU as the device. Device code is written in what are called kernels, functions that contain the data-parallel code to be executed on the GPU. Aside from more recent advancements in the release of CUDA 4.0, namely Unified Virtual Addressing (UVA), host code and device code have separate memory spaces. Thus, it is necessary to transfer data being computed on the device over from the host with functions provided by the CUDA API. Typically, a program will allocate memory on the host (*malloc()*, for example), allocate memory on the device (*cudaMalloc()*), transfer the data stored in the host's allocated memory to the device's allocated memory (via *cudaMemcpy()*), and after computation on the device, transfer the results back to the host (also with *cudaMemcpy()*).

B. Thread Organization

In order to understand how the kernel code is executed in parallel, we now investigate the organization of threads. This insight will lead to discussion of a crucial factor in an efficient

device code implementation, memory access efficiency. Kernels execute in parallel across a set of threads, when actually writing kernel code threads can be determined by built-in variables. Threads are organized into two larger groups:

- Thread blocks are a set of threads that can cooperate to complete tasks and share some resources such as shared memory private to the thread block.
- A grid is a set of thread blocks which are independent from one another and able to be executed in parallel.

C. Types of Device Memory

These form a hierarchy of thread groups each with access to varying memory types. The table below illustrates which memory each thread group can access in device code. It is important to note that data can only be transferred to/from the host and device using global and constant memories.

Memory Type	Thread Group Access	R/W
Registers	Thread	R/W
Local Memory	Thread	R/W
Shared Memory	Block	R/W
Global Memory	Grid	R/W
Constant Memory	Grid	R

Table 1 Thread Groups Access to Device Memory

Memory usage is an important consideration when developing CUDA applications. The goal should be to maximize the use of on-chip memories such as registers and shared memory. Furthermore, two low bandwidth transfers can become a major bottleneck if performed excessively, that is transferring data between host and device as well as global memory and device. This is due to data transfers between on-chip memory and off-chip memory. Because of the overhead involved in each transfer, better performance can achieved by packing many small transfers into a single large transfer [5]. We will now briefly survey some general properties for each memory type in Table 1.

Registers are local to each thread and hold automatic variables other than arrays. Registers provide very high speed memory and can be used to hold frequently accessed variables. Local memory is visible only to a single thread. Automatic variables may be placed in local memory if they are too large for the thread's registers or if no more registers are available. Local memory is stored in external DRAM so it has the same high latency and low bandwidth as global memory accesses. Shared memory is private to each thread block (shared among threads in the block) and occupies storage from creation to termination. Therefore, it is stored on chip in SRAM and is not constrained by slow off-chip bandwidth.

Global memory is the largest and slowest memory space. Being stored on external DRAM, it can be accessed by different thread blocks in different grids. A parallel algorithm should be carefully analyzed to identify an alternative to global memory when possible. Constant memory is read-only memory stored in external DRAM. However, constant memory supports short-latency, high-bandwidth access when all threads simultaneously access the same memory location.

D. Parallelization

GPUs are well suited for problems that exhibit data parallelism with high arithmetic intensity. When analyzing a program's sections for CUDA implementation, it is important to note any data parellelism. These sections possess many arithmetic operations that can safely be performed in parallel. Generally, problems that are highly parallel are large problems that need to be decomposed into many small problems. For example, a code section may be found that is favorable for parallelization. It is then necessary to find parts of the algorithm that can be computed idependently in parallel, these can grouped into thread blocks.

III. Classic Example: Matrix Multiplication

A classic example that exhibits a rich amount of data parallelism is matrix multiplication. In this section, we explore a basic sequential and parallel implementation.

A. Problem Description

It is presumed the reader has studied matrix multiplication. This section serves as a brief review. Matrix multiplication is a binary operation that produces a product matrix C by taking the dot product between two input matrices, matrix A and matrix B. The product of matrices A and B is defined as (using Einstein summation):

$$c_{ik} = a_{ij} b_{jk} \tag{1}$$

where j is summed over all possible values of i and k [11].

In order for matrix multiplication to be valid, the dimensions of the input matrices must satisfy:

$$(n \times m)(m \times p) = (n \times p) \tag{2}$$

More verbally, it can be seen the element in the product matrix can be found multiplying the elements of the same row from the first input matrix by the corresponding elements of the same column of the second matrix and summing the results.

B. Simple Sequential Implementation

The last paragraph of the preceding section describes an algorithm that can be used to calculate the product matrix. For simplicity, we assume in the sequential and parallel implementation an equal width of input matrices. This ensures the dimensions validity requirement is met. The algorithm below is a straightforward translation of the previous paragraph into C:

```
__host__ void HostMatrixMul
(float **A, float **B, float **C, int Width)
{
   int i = 0;
   int j = 0;
   int k = 0;

   for(i = 0; i < Width; i++)
        for(j = 0; j < Width; j++)
            for(k = 0; k < Width; k++)
                C[i][j] +=  A[i][k] * B[k][j];
  }
```

Fig. 1 Sequential Matrix Multiplication

If the reader is inexperienced in CUDA programming, the __host__ qualifier prefixed to the return type is unfamiliar. This indicates the function is executed on the host and called by the host. In this case, __host__ is not actually needed; this is the default for a function with no qualifier. In Fig. 2 we see __global__, denoting a CUDA kernel function that is executed on the device and only callable from the host.

C. Simple Parallel CUDA Implementation

Inspecting the actual computation in Fig. 1, it can be concluded this is an excellent candidate for parallelization. Computing the dot product for each element is completely independent of other elements.

```
__global__ void MatrixMulBlocksKernel
(float *Md, float *Nd, float *Pd, int Width)
{
 // Calculate the row index of the Pd element and M
 int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;

 // Calculate the column index of Pd and N
 int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

 float Pvalue = 0;
 for (int k = 0; k < Width; ++k)
      Pvalue += Md[Row * Width + k] *
                Nd[k * Width + Col];

      Pd[Row * Width + Col] = Pvalue;
}
```

Fig. 2 Parallel Matrix Multiplication Kernel [6]

Fig. 2 is the kernel formulated from the sequential code sample in Fig. 1. Recall, the kernel specifies the code to be executed by all threads during a parallel phase. Taking the perspective of a single thread, let's review the functionality of the code in Fig. 2. *Md* is the first input matrix, *Nd* the second input matrix, and *Pd* the product matrix. *TILE_WIDTH* is a macro specifying the dimensions of the matrix (*TILE_WIDTH* × *TILE_WIDTH*).

To reach our desired single thread perspective we must determine which current thread we are executing. The thread groupings outlined earlier can be accessed using the built-in variables *blockIdx* and *threadIdx*. They are built-in in that they are pre-supplied with the information specific to each thread when accessed by that thread. Note that another built-in variable *blockDim* could have been used to determine the dimensions of the thread blocks, but for simplicity *TILE_WIDTH* is used.

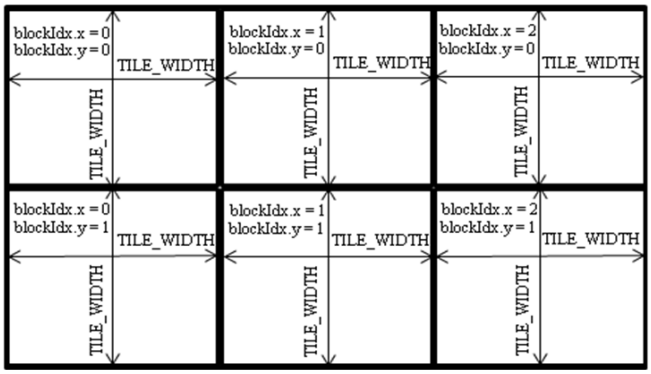

Fig. 3 Thread Blocks Organized in a Grid

Fig. 3 is a grid of thread blocks. The dimensions of thread blocks in this figure can be determined by built-in variable *gridDim*. The dimensions of each individual block can be determined with *blockDim*. These built-in variables are actually structs and can supply the x and y dimensions, for example *blockDim.x*. Using this information, we return to the critical calculation of our current working thread. To obtain our thread's row, the thread's block vertical offset (*blockIdx.y*) is multiplied times the height of each block (*TILE_WIDTH*). This places our calculation in the first row of the thread's block. Now, to arrive at the correct row, we add the offset from this first row, the thread's y value in respect to its containing block, *threadIdx.y*. *Col* is calculated similarly but in the x direction.

A key difference between Fig. 1 and Fig. 2 is the reduction of loops in Fig. 2. Notice that in the sequential version the rows and columns must be iterated through. However, in the CUDA kernel, threads are executed in parallel and the row and column are immediately available. The final step is to use the row and column to actually perform the dot product. The for loop in Fig. 2 performs the calculation of the dot product. The indexing of the *Md* and *Nd* arrays is a similar concept to how the row and column were calculated. Here we iterate *k* across the dimensions (passed in as *TILE_WIDTH*) and sum the values from the corresponding row and column for the current product element being calculated. A final observation is to keep in mind this is done by each thread. Therefore, we can simply set the *Pd* value that specific thread was responsible for at the end of the kernel.

D. Performance Comparison and Conclusion

This example is relatively simple. It is intended to ease a reader into an understanding of the CUDA programming model by illustrating the transformation of a sequential algorithm into its parallel counterpart. The SPMD style of programming is exposed as we wrote a small program for one thread that is instanced and executed over potentially thousands of threads in parallel.

Our main motivation for parallelization was to achieve speedup over the limited throughput of sequential execution.

The performance results for matrices of increasing sizes are provided in Table 2.

Matrix Dimensions	Sequential Execution Time (ms)	Parallel Execution Time (ms)	Speed up
250 x 250	660	1.3	508
500 x 500	2610	2.0	1305
750 x 750	9110	3.6	2531
1000 x 1000	26260	5.7	4607
1500 x 1500	96520	11.9	8111

Table 2 Execution Times for Matrix Multiplication Examples

IV. SSD Simulation Example

The matrix multiplication example introduced some of the fundamental concepts needed to understand a CUDA kernel. Building on this foundation, in this section a more complicated program is discussed. The problem is to estimate the steady-state distribution for large networks by the Kolmogorov-Smirnov (KS) test based on simulation of Monte Carlo methods. For the scope of this report, our focus is on the structure of the problem and how we can parallelize the sequential version into a CUDA kernel rather than a strong understanding of the mathematical theory behind the code.

A. Sequential Implementation

In contrast to the matrix multiplication example, here we encounter a scenario where a large component of the problem is ideal for parallelization, but a small segment is more appropriate for execution on the CPU. Let's examine the code structure. Once again, neglecting the mathematical reasoning on which it is based.

Variable Name	Value
num_genes	Input to program by user
nstates	Array of random states input by user
p	Input to program by user
num_sts	2^{num_genes}
total_T	num_sts * 30
total_N	num_sts * 50
iters	total_T + total_N
iter_ks	3 * num_sts
smpl_int	5
running_sts	2-D array of states initialized to zero

Table 3 Undeclared Variables Used in Sequential SSD Example

Table 3 lists the variables that are not declared in the example and how their value is calculated. As shown in the table, there are three inputs to the function: *num_genes* (number of genes simulated), *nstates* (array representing a random state transition table), and *p* (perturbation probability). These variables are assumed to be calculated outside the loop in Fig. 4, which is the portion suitable for parallelization:

```
for (unsigned int indr = 0; indr < iter_ks; ++indr)
{

 unsigned long curr_st = 0 + rand() % (num_sts - 1);

  for (unsigned indi = 0; indi < iters; ++indi)
  {
    unsigned long int next_st = curr_st;
    bool count_pert = 0;

  // Toggle each bit (num_genes many) for a random
  //  next state
  for (unsigned int indg = 0; indg < num_genes;
       ++indg)
  {
    double prand = rand_frac(); // between 0 and 1

    if (prand <= p)
    {
      count_pert = 1;
      next_st ^= 1 << indg; // toggle indg bit
    }
  }

  // If next_st wasn't altered change it here
  if (!count_pert)
    next_st = nstates[curr_st];

  if (indi % smpl_int == 0)
    if (indi > total_T)
      if (indi <= (total_T + total_N/2))
        running_sts[0][next_st] =
          running_sts[0][next_st] + 1;
      else
        running_sts[1][next_st] =
          running_sts[1][next_st] + 1;

  curr_st = next_st;
  }
}
```

Fig. 4 Sequential SSD Example to be Parallelized

B. Deriving CUDA Parallelism Structure

To convert this loop to a CUDA kernel, we naturally first inspect the two "large" outer loops. What is important for CUDA kernel implementation is to note any dependencies among variables in the two loops. The goal is to map iterations of the larger loop to CUDA threads. The outer loop will iterate *iter_ks* times and the inside loop *iters* times. Referencing Table 3 reveals *iters* will be significantly greater than *iter_ks*.

Initially, this revelation may seem potentially worrisome. We would like our outer loop to incur more iterations than the inner loop. However, some techniques are available to help us cope with the situation. A first option to consider is loop fission, the splitting of two loops. If parts of the loop body are independent of other parts they can be executed in a loop and then the remaining parts executed in a proceeding loop. Once separated, the two loops could be implemented in two kernels and executed in sequence. This technique would not sacrifice any parallelism.

Another option is loop interchange, to interchange the inner and outer loops and then map each thread to an iteration of the new larger outer loop. Intuitively, it may be uncomforting to perform such a maneuvering of code. Nevertheless, if both levels of two loops are independent of each other their relative order of execution is inconsequential [4].

C. Parallel CUDA Implementation

Although the CUDA C code in Fig. 4 may have appeared daunting to follow at first glance, techniques from the preceding section suggest key properties to recognize in the code structure.

```
__global__ void CudaSSD(
  unsigned int *glob_running_sts1,
  unsigned int *glob_running_sts2,
  unsigned long int *nstates,
  float *rndFloats,
  unsigned int *drndNums)
{
  /* Kernel Indexing */
  int Tid = threadIdx.x +
            blockDim.x*threadIdx.y +
           (blockIdx.x*blockDim.x*blockDim.y) +
           (blockIdx.y*blockDim.x
             *blockDim.y*gridDim.x);

  unsigned long int curr_st =
          0 + drndNums[Tid] % (NUM_STS - 1);

  unsigned long int next_st = curr_st;

  bool count_pert = 0;

/* Toggle each bit (num_genes many) for a
     random next state */

  for (unsigned int indg = 0;
       indg < NUM_GENES;
       ++indg)
  {
    // rndFloats[Tid] here is "prand"
    if (rndFloats[Tid] <= P)
    {
      count_pert = 1;
      // toggle indg bit (for next_st)
      next_st ^= 1 << indg;
    }
  }

/* If next_st wasn't altered change it here */
  if (!count_pert)
    next_st = nstates[curr_st];

  unsigned int tidMIters = Tid % ITERS;

  if (tidMIters % SMPL_INT == 0
      && tidMIters > TOTAL_T)
  {
    if (tidMIters <= (TOTAL_T + TOTAL_N/2))
      atomicAdd(&glob_running_sts1[next_st], 1);
    else
      atomicAdd(&glob_running_sts2[next_st], 1);
  }

  __syncthreads();

  curr_st = next_st;
}
```

Fig. 5 SSD Example CUDA Kernel

We begin analysis of this kernel by accounting for the variables in Table 3. All variables other than *nstates* and *running_sts* can be calculated prior to any kernel invocation. In fact, in this implementation, the user assigns an integer to a *NUM_GENES* macro and, in turn, the other variables implemented as macros are determined. Since macros are in the code are textually substituted by the preprocessor this allows consistent values between the host code and device code. Recall that *nstates* is a table of random state transitions. This is simply an array whose length is the number of states (*num_sts*) filled with integers in the range [0, *num_sts*-1]. In order to simplify the kernel code, *running_sts* has been split into two separate arrays as it only occupied two rows before. The calculation of a thread index (*Tid*) uses the built-in variables discussed in the matrix multiplication example. It is left to the reader to verify the procedure's correctness.

Now we provide an explanation for the kernel parameters:

1) *running_sts*: The two *running_sts* arrays passed in are empty. They are assigned values during the execution of the kernel and transferred back to the host for post-kernel processing. This is the only data transferred to the host from the device.

2) *nstates*: *nstates* is transferred to the kernel for kernel computation. It is only read from the kernel and does not need to be transferred back to the host.

3) *rndFloat and drndNums*: To maintain consistency with the sequential version, an array of random floats and random integers are filled on the host and transferred to the kernel when random numbers are needed.

After evaluating the two outlying loops in Fig. 4 it was concluded the inner loop based on *iters* iterations would far exceed the number of iterations performed by the outer *iter_ks* loop. The implementation in Fig. 5 is based on the observation that the two loops in question are independent. Indeed, the only statement, assignment of *curr_st*, in Fig. 4 prior to the *iters* loop can simply be converted to an assignment of a random number to an automatic variable (register) for each thread. The for loop with *NUM_GENES* iterations is left intact because the number of genes is typically not large enough to warrant parallelization.

The final series of if statements references *indi* often. In Fig. 5, *indi* is replaced by the modulo operation of the thread id and *ITERS*, the latter being the bound on the loop in the sequential version. This frequency of access can be taken advantage of by placing the result in a register that can be quickly accessed for each reference. In addition, we want to reduce branching. A simple alteration is to merge the first and second if statement by a logical and of the branch conditions as shown in Fig. 5.

The remaining topic of the kernel is the *atomicAdd()* function and the *__syncthreads()* call. An atomic operation protects the sequence of operations on a variable. We are

aware *glob_running_sts1* and *glob_running_sts2* are being executed by multiple threads simultaneously. A more subtle issue is that of these threads attempting to modify a shared memory location. The operation desired is to simply increment the indexed *glob_running_sts* value by 1. To further understand the problem, assume the value in *glob_running_sts1* is 15 prior to the call and the following order of operations occurs:

- Thread A reads the value in *glob_running_sts1*
- Thread B reads the value in *glob_running_sts1*
- Thread A adds 1 to the value it read
- Thread B adds 1 to the value it read
- Thread A writes the result back to *glob_running_sts1*
- Thread B writes the result back to *glob_running_sts1*

Two threads were directed to increment the value by 1. The expected answer would be 17. However, this interleaved arrangement of operations results in a final value of 16. The problem is that threads are scheduled unpredictably and can compute an incorrect result. In order to remedy this situation an atomic operation is needed. An atomic operation guarantees uninterrupted access to a memory location by a thread even when thousands of threads are competing for access. *atomicAdd(&glob_running_sts1[next_st], 1)* provides such protection for the element incremented in the *glob_running_sts1* array. Finally, *__syncthreads()* guarantees every thread in the block has completed instructions prior to the call and before the next instruction is executed on any thread [9].

D. Code Remaining on Host for Execution

It was mentioned a portion of the computation code would not benefit from parallelization. This code is briefly examined to understand why it was left to be performed on the host.

```
double *ssd = (double *)calloc(NUM_STS,
                sizeof(double));

for (unsigned int i = 0; i < NUM_STS-1; ++i)
     ssd[i] = 0.0;

unsigned long running_sts_sum = 1;

for (unsigned i = 0; i < NUM_STS; ++i)
     running_sts_sum += running_sts2[i];
for (unsigned i = 0; i < NUM_STS; ++i)
     ssd[i] = running_sts2[i] /
              (double)running_sts_sum;

double kss = 0.0;
for (unsigned i = 0; i < NUM_STS; ++i)
// set kss to max temp_result value determined
{
     double temp_result =
      abs(running_sts1[i]/running_sts_sum - ssd[i]);

     if (temp_result > kss)
         kss = temp_result;
}
```

Fig. 6 SSD Example Code Left for Execution on CPU

Overlooking the objective of this code and just evaluating the structure of the instructions reveals no potential candidate for parallelization. The loops all iterate *num_sts* times which is not by any means considered a significant number of repetitions. This limits the degree of parallelization possible. Moreover, the computations are relatively simple and the GPU is best suited for compute intensive tasks. In summary, this simple code is not appropriate for execution on the device and performs better on the host.

E. Performance Comparison and Conclusion

Throughout the implementation process of this example, two important lessons were learned. First, keep in mind that the host and device have separate memory spaces. Consequentially, libraries such as the C++ Standard Template Library (STL) will not be available in the kernel code. As a side note, a reader familiar with the STL may be interested in researching Thrust. Thrust is a CUDA library of parallel algorithms with an interface resembling the STL [1]. Secondly, consider alternative approaches to converting a problem into a CUDA kernel. Particularly, the thread and memory limit are two factors to keep in mind. For instance, try to recognize situations in which techniques such as loop fission or loop interchange can reduce thread consumption on a single kernel.

The thread and memory issue was a particular challenge to overcome outside the development of the kernel. For example, the total number of threads launched is *iters* × *iter_ks*. Referring to Table 3, at *12* genes *iters* is equal to 327,680 and *iter_ks* is equal to 12,288, their product is 4,026,531,840. This is an enormous number of threads to attempt to launch and far exceeds the 67,107,840 thread hardware limit on the NVIDIA GeForce GTX 560 on which the code is executed. The key observation to circumvent this challenge is that *running_sts1* and *running_sts2* are the only data transferred back to the host from the kernel. These results can be accumulated in a host array as the kernel is invoked multiple times with each iteration returning a new subset of values. Each time the kernel is launched with the maximum number of threads possible until all the needed threads are accounted for. After which, the collective *running_sts1* and *running_sts2* arrays on the host are processed by the code in Fig. 6.

Number of Genes	Sequential Execution Time (ms)	Parallel Execution Time (ms)	Speed up
5	3780	1119.4	3.38
6	8420	1439.6	5.85
7	18860	1525.5	12.36
8	45020	1807.0	24.91
9	92380	5814.8	15.89
10	203280	22598.8	9.00
11	442310	95544.8	4.63
12	955260	362936.0	2.63

Table 4 Execution Times for Simulate SSD Example

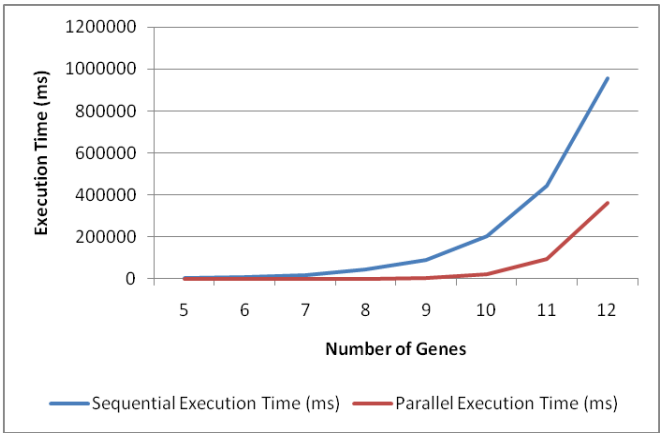

Fig. 7 Execution Time vs. Number of Genes for Simulate SSD Example

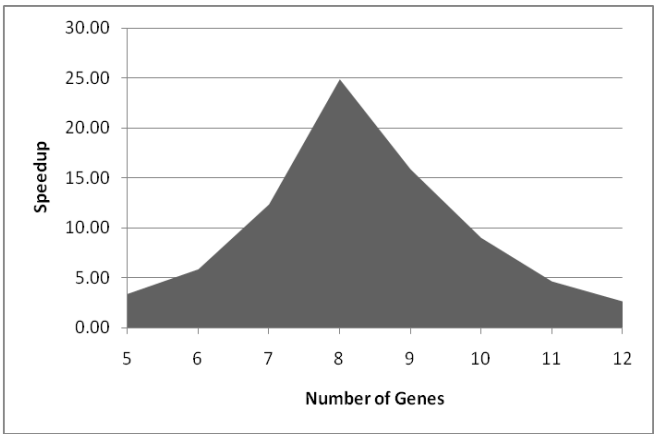

Fig. 8 Speedup vs. Number of Genes for Simulate SSD Example

V. Random Logic Simulation Example

This example is based on the work found in [10].

A. Problem Description

Today, electronic designs are more complex than ever before and verification of such circuits consumes a substantial portion of the overall design cycle. A GPU-based parallelization of logic simulation for electronic designs could have a profound impact on industry in both financial savings and a reduction in time-to-market. Verification is necessary to ensure the design behaves according to specifications.

The purpose of this example application is to develop a random logic simulator that can be easily parallelized and implemented in CUDA to achieve a speedup over existing methods. Logic simulation involves two major phases, compilation and simulation. Compilation will take the circuit specified in a file and convert it into data structures and a format appropriate for parallel implementation in a CUDA kernel. This phase is executed on the host and consists of a novel method discussed in [10] called levelization that constructs levels of gates that can simulated independently of one another. Simulation is executed on the kernel and similar to previous examples; it will harness the thousands of available threads to achieve speedup.

B. Background

Two common types of logic simulation that are used in industry include Cycle-Based Simulation (CBS) and Event-Based Simulation (EBS). In CBS, each step of simulation is determined at compile time. In EBS, gates are simulated only when their inputs change. Our design employs CBS as this is less complicated than EBS and better adapted to parallelization. The tradeoff for convenience of CBS is that EBS is more efficient because it does not simulate unnecessary gates.

Following the design choice in [10] the chosen design representation for the logic circuit is an And-Inverter Graph (AIG), in AIGER [2] format. An AIG is a directed, acyclic graph that efficiently represents a boolean function. Every gate in an AIG is expressed in terms of two-input AND nodes (gates) and inverters. Single input nodes represent memory elements such as latches. AIGs are widely used in industry for logic synthesis and verification and have been studied extensively since the 1950s [3].

AIGER is the file format used that contains the structure of the AIG. AIGER file formats are available in a binary (.aig extension) and ASCII form (.aag extension). We avoid use of the AIGER library so our format of choice is the ASCII version. According to [2], the first line in an AIGER ASCII format file is specified as follows:

```
aag M I L O A
```

Each identifier is replaced with the non-negative integer specifying the following attributes: M is the maximum variable index, I is the number of inputs, L is the number of latches, O is the number of outputs, and A is the number of AND gates. Each variable index n holds a boolean value. An odd numbered index, n+1 represents the value of n inverted.

C. Overview and User Interface

More internal details are revealed with this implementation than preceding examples in order to guide the programmer in converting the current simulation algorithm into a parallel version. This initial sequential version simulates each logic gate sequentially one at a time. However, an effort has been made to setup the algorithm and data structures available to ease parallel implementation.

From a user's point of view, they can declare an object of *Aiger* type and pass to its constructor the path to an ASCII AIGER file. Internally, the class automatically performs the compilation phase. The user can now call the method for simulation repeatedly for the number of desired simulation cycles. *DisplayAdjacencyList()* is available if the user desires to see the adjacency lists built for each AND node in the AIG.

D. Levelization and Clustering

Our goal in the kernel is to simulate the circuit level-by-level establishing a dependency relation between the levels. Simulation starts at the gates in the lowest level and progresses upward to circuit outputs. Levelization determines which gates can be independently simulated together, that is, they do not depend on each other and their inputs come from outputs on a lower level. Latches and circuit inputs are not included in levelization, they can be considered to be in level 0. We can now define the level of a gate as the largest distance either to a circuit input or memory element in the design [10].

After levelization, another circuit partitioning technique is clustering. Clustering aims to group only the combinational and sequential elements needed for a particular circuit into a cluster. Each cluster can then be simulated independently of other clusters and each level in the cluster is able to be simulated independently of other levels. This greatly enhances the degree of parallelization possible.

E. Sequential Implementation

The main data structures are the *and2*, *latch*, and *Aiger* classes. *and2* represents a simple AND gate that stores input variable indices, an output variable index, an output value (determined in simulation), and level (determined in levelization). In addition, since it is a node in an AIG it also stores an adjacency list (determined in levelization). The *latch* class represents a latch and simply stores an input and output variable index and an output value. The circuit is represented in a class called *Aiger*. Fig. 9 is a UML description of the *Aiger* class:

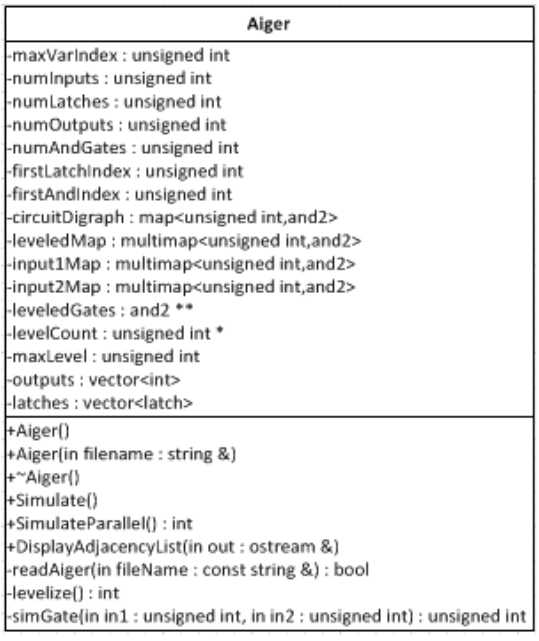

Fig. 9 Aiger UML Class Description

Many of the class member's purpose are self-explanatory and highlighted in source code comments. The relevant data structures for parallel implementation are especially *leveledGates*, *levelCount*, and the variable *maxLevel*. In particular, *leveledGates* was created specifically for parallel implementation since the previous container of *and2* objects, *circuitDigraph*, is implemented in the STL and unavailable on a CUDA kernel. *leveledGates* is a 2-D array of *and2* gates. The first dimension is the level of the gate and the second dimension is the index of each gate on that level. This arrangement was chosen to efficiently map the results from levelization onto kernel threads in an efficient manner.

We now outline the private methods of the *Aiger* class:

- *readAiger()* is a trivial method that reads the required data from an AIGER file into the class member variables for use in levelization and simulation.
- *levelize()* performs the "horizontal slicing" of the AIG. The implementation for levelization is contained within this method. After levelization, the class is ready for simulation.
- *simGate()* simply returns a bitwise AND of its two unsigned integer parameters.

After highlighting the classes used and methods implemented we can now examine high-level pseudocode for the current sequential simulation method (*SimulateParallel()*). This general algorithm is provided to aid in understanding the C++ implementation in the source code file.

```
generate numInputs random inputs

for i = 0 to maximum level of circuit
   for i = 0 to current level gate count
      if first input = 0 then
         first input value = 0
      else if first input = 1 then
         first input value = 1
      else if first input < first latch index
         first input value = matching index from circuit input
      else if first input < first and index
         first input value = matching index from latch
      else
         first input value = matching index output from AND
                             gate
      endif
      if first input variable index is odd then
         invert first input value
      endif

      if second input = 0 then
         second input value = 0
      else if second input = 1 then
         second input value = 1
      else if second input < first latch index
         second input value = matching index from circuit
                              input
      else if second input < first and index
         second input value = matching index from latch
```

```
        else
          second input value = matching index output from
                               AND gate
        endif
        if second input variable index is odd then
          invert second input value
        endif

        simulate gate using the two input values
      endfor
    endfor

    update sequential logic based on current circuit inputs and
      updated gate values
```

Fig. 10 Pseudocode for Sequential Simulation Algorithm

F. Conclusion

Another work, [8], is also based on [10]. Although much greater performance was anticipated from the parallel simulation method, the result was exceptionally negative speedup. This provides evidence that a poor CUDA implementation can have an adverse effect on performance. In contrast, that work is based on the same principles as [10] whose experimental results indicate a fast, efficient parallel logic simulator is attainable using the levelization and clustering techniques.

VI. Conclusion

In this paper, we have introduced the NVIDIA CUDA programming model and software platform for the GPU. Two applications of CUDA were explored to demonstrate the simplistic extension to the C and C++ programming languages offered by CUDA. In addition, the matrix multiplication and Simulate SSD kernels suggested that positive speedups in performance are not too difficult to attain. The Random Logic Simulator example presents a practical application of exploiting the massively parallel GPU architecture to solve large-scale problems in modern industry.

References

[1] NVIDIA. (2011, July) GeForce GTX 285. [Online]. HYPERLINK "http://www.nvidia.com/object/product_geforce_gtx_285_us.html" http://www.nvidia.com/object/product_geforce_gtx_285_us.html

[2] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide," NVIDIA, Santa Clara, Documentation 2010.

[3] Nathan Bell and Jared Hoberock. (2011, July) Thrust. [Online]. HYPERLINK "http://code.google.com/p/thrust/" http://code.google.com/p/thrust/

[4] Armin Biere, "The AIGER And-Inverter Graph (AIG) Format," Johannes Kepler University, Linz, Documentation 2007.

[5] Leo Hellerman, "A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits," *IEEE Transactions on Electronic Computers*, vol. EC-12, no. 3, pp. 198-223, June 1963.

[6] David B Kirk and Wen-mei W Hwu, *Programming Massively Parallel Processors*. Burlington, United States of America: Morgan Kaufmann, 2010.

[7] David A Patterson, John L Hennessy, John Nickolls, and David Kirk, *Computer Organization and Design*, 4th ed. Burlington, United States of America: Morgan Kaufmann, 2009.

[8] William Pence, "GPU-Based Parallel Algorithm for Random Logic Simulation," University of South Florida, Tampa, Research Report 2011.

[9] Jason Sanders and Edward Kandrot, *CUDA By Example An Introduction to General-Purpose Programming*, 1st ed. Boston, United States of America: Pearson Education, Inc., 2011.

[10] Alper Sen, Baris Aksanli, Murat Bozkurt, and Melih Mart, "Parallel Cycle Based Logic Simulation using Graphics Processing Units," in *2010 Ninth International Symposium on Parallel and Distributed Computing (ISPDC)*, Istanbul, 2010, pp. 71-78.

[11] Eric W Weisstein. (2006, March) MathWorld--A Wolfram Web Resource. [Online]. HYPERLINK "http://mathworld.wolfram.com/MatrixMultiplication.html" http://mathworld.wolfram.com/MatrixMultiplication.html