Highly Parallel Implementation of NeuroJet using Graphics Processing Units

A Technical Report
in STS 4020

Presented to

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment
Of the Requirements for the Degree

Bachelor of Science in Computer Engineering

by

Sang-Ha Lee

May 3, 2010

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for the Undergraduate Thesis Project.

____________________________________ ____________

Sang-Ha Lee Date

Approved by:

____________________________________ ____________

Kevin Skadron, Department of Computer Science Date

Highly Parallel Implementation of NeuroJet using GPUs

Sang-Ha Lee and Kevin Skadron

Department of Computer Science, University of Virginia
sl4ge@virginia.edu, skadron@cs.virginia.edu

Abstract. This paper presents the parallel implementation of NeuroJet, a neural network simulation program, using GPUs. To help researchers study different correlation among applications, programming models, and underlying hardware architectures, NeuroJet is examined as a case study to provide insights and findings specific to GPU programming. We demonstrate the importance of programmer effort through our parallel implementation of NeuroJet. Our study also shows that GPUs provide substantial performance improvement up to 500,000X speedup, albeit in expense of readability and maintainability of the code, which is not desirable in certain cases.

Keywords: GPU, CUDA, Octave, NeuroJet

1 Introduction

As processors have evolved rapidly towards multi- and many-core architectures, software that leverages such hardware has been increasingly emphasized by both hardware and software researchers. Developing applications for parallel architectures require non-intuitive techniques and approaches, which mainly includes understanding data structures, memory access patterns, and algorithms. Some of these techniques and approaches are generally applicable, where others are specific to certain types of applications.

In this paper, we accelerate NeuroJet, a neural network simulation program. From this case study, we attempt to contribute general knowledge, insights, and lessons learned about multi- and many-core programming. We first provide some background information and explore the application and platforms relevant to the work. Then, NeuroJet's overall structure and specific algorithms and data structures are summarized, and subsequently profiled to guide parallel implementation strategy and optimization on GPU. Further discussion on parallelization and optimization technique follows, with performance results and analysis on performance bottlenecks. Finally, we discuss consequential insights and discoveries attained during the implementation, optimization, and analysis that may deliver valuable implications for forthcoming multicore architectures and programming models.

We found that developing applications for parallel architectures requires extensive programmer effort on implementing and optimizing the application. In particular, optimizations specific to the target platform are essential for the best performance. However, we also learned that pushing limits on the performance is costly in optimizing the program and reduces readability thereby.

2 Application/Platforms

The version of NeuroJet we obtained and utilized for analysis is written in MATALB-compatible Octave script. This script is used as a baseline for our study, and later accelerated on NVIDIA GPU using CUDA API. Thus, it is worthwhile to discuss the background and relevant information pertaining to these hardware and software platforms. However, our intention is not to cover details of these platforms or the science behind the application beyond our scope.

2.1 NeuroJet

NeuroJet simulates the hippocampal region of brain at high speeds while allowing users to easily sweep parameters. It models repeated steps of brain's learning and memory tracing capability by applying established formulas that calculate and update the excitation and the firing patterns of neurons [8]. In simpler words, a set of formula is applied to each neuron to project the activities of neurons. The application allows users to run the simulation with desired number of neurons and outputs the firing pattern data that can be graphically displayed using MATLAB. Octave script provides ease of programming for users while preserving the biological concept visible in the script itself.

In order to efficiently parallelize applications on GPUs, programmers must know what data structures the application is using and what memory access patterns it is implementing. Data structure optimized for one architecture may not provide the most optimized access pattern for the other, thus it may need to be rearranged accordingly in order for certain architectures to provide most optimized memory transactions. Such fact holds true for GPU that coalesced memory access provides improvement over non-coalesced accesses. Therefore, knowing how data is laid out in memory (data structure) and how it is being accessed (memory access pattern) is important when parallelizing an application.

Note that details of the algorithm involve extensive knowledge of neuroscience that may distract readers from focusing on characteristics of the application, and are omitted for that purpose. Only the knowledge necessary to understand application's memory access pattern and data layout is covered. Figure 1 [10] shows the schematic representation of simulated network. Each neuron is connected to multiple other neurons and this connection, or synapse, has weight value associated with it, representing the

strength of the connection. Connected neurons can excite each other and the excitation level may exceed a certain threshold value and cause neurons to fire (transmit chemicals through synapse).

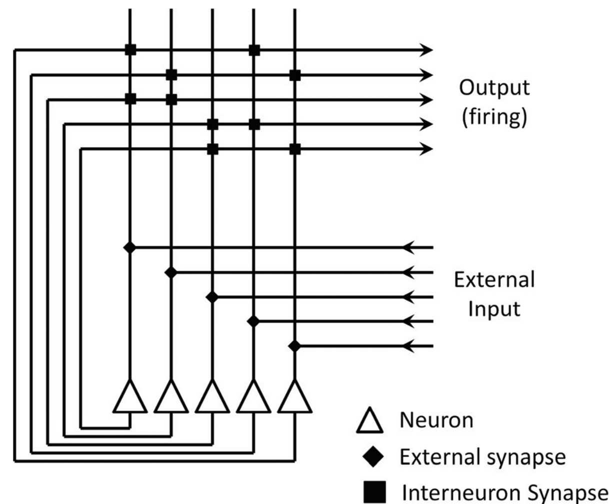

Fig. 1. Interneuron synapses are represented with dark squares. Excitation induced by interneuron synapses along with synapses from external inputs determines the output of the neuron.

The connection and interaction of neurons are largely represented in NeuroJet with four matrices.

- **Connections.** Represents connections among neurons (interneuron synapses in Figure 1). Neurons have random and sparse connectivity, having only 10% connectivity. Thus, the rows of the matrix represent neurons and the columns of the matrix represent that neuron's connection to other neurons. For instance, if 1000 neurons are simulated, then the matrix is 1000×100 in size, representing 1000 neurons and each neuron's 100 connections with other neurons. The matrix is initialized with random values to represent random connectivity and remain unchanged for the duration of simulation.
- **Weights.** Represents the synaptic weights of corresponding connections represented in Connections matrix. Therefore, the size of Weights matrix is equivalent to that of Connections matrix. The weight values are updated in each time step according to the weight-updating formula. There are two other weight vectors, namely FBWeights (feedback) and FFWeights (feed-forward) that contribute to the inhibitory force on exciting a neuron.

- **Z.** Represents the firing of each neuron in each time step. Thus, the rows represent neurons and columns represent binary value of the firing of each neuron in each time step. If 1000 neurons are in simulation and simulation progresses to 50 time steps, then the size of Z matrix is 1000×50.
- **Inputs.** Represents the external inputs to the system in each time step (external synapses in Figure 1). These inputs are purposely setup to observe the reaction to inputs. As in Z matrix, the rows represent neurons and the columns represent external input to each neuron in each time step. Input values are binary as well.

There are a few other matrices and vectors, but they do not contribute in computation as heavily as the above four. Octave stores matrices in column-major order as in MATLAB, which, as we will see later in this paper, eliminates the effort to coalesce GPU global memory accesses. The algorithm involved in calculating the excitation and updating the synaptic weights is not computationally intensive, yet involves lots of memory accesses.

2.2 Octave

Syntactically, Octave is mostly compatible to MATLAB with few minor exceptions such as support for C-based operators like ++, --, and += and use of both single and double quotes for defining strings in Octave compared to MATLAB. Thus, Octave allows users to write scripts for both MATLAB and Octave without much overhead of conversion from one to the other. However, the support for Just-In-Time (JIT) compiler in MATLAB makes a significant performance difference when for-loops are extensively used instead of vectorization [9]. The performance evaluation conducted by Chaves, et al. [3] shows that there is some performance difference between MATLAB and Octave in regular built-in functions as well. Octave, however, still provides an alternative to MATLAB without compromising productivity and with acceptable performance. Its open source distribution is also a merit that allows more accessibility. Octave version 3.0.1 is used for this work.

2.3 CUDA

CUDA [5] provides an API similar to C with minor extensions and allows programmers to explicitly move data between main memory space and GPU global memory space and perform computation on GPU that can launch a large number of concurrent threads. A code that runs on GPU is called a kernel, which is distinguishable from a code that runs on CPU. Kernels allow programmers to simplify the parallel implementation by writing a code for a single thread that is invoked by all threads. Hardware manages thread creation and scheduling, thus programmers do not need to be concerned about thread management. We use CUDA version 2.1 for this work.

One of the most important features of CUDA is that groups of 32 threads called *warps* execute in a lockstep, and it is important to avoid divergent path among those 32 threads in a warp since only threads on the same branch path can run simultaneously. Also, memory transaction can be optimized for *half-warps* (either first or second half of a warp) when threads in a half-warp simultaneously access memory segments in sequence (called *coalescing*). Thus, memory access by threads in a half-warp results in a single memory transaction at best case, and 16 different transactions at worst case.

Currently, NVIDIA GPUs support CUDA API. The NVIDIA Tesla C1060 card is used for this work and it has 30 streaming multiprocessors (SMs), each of which consists of 8 streaming processors (SPs), for a total of 240 SPs. 8 SPs in each SM shares 16KB shared memory, and each SP has a floating point ALU that can perform integer operations as well. The processor frequency is 1.3GHz, and it has 4GB of GDDR3 memory. Data between the main board and GPU board are exchanged via PCIe×16 bus [6], [7].

3 Parallel Implementation

3.1 Profiling

Profiling NeuroJet is done by decomposing the application into several steps and measuring execution time consumed in each step. NeuroJet is composed largely of 4 steps, namely Setup, CalcExcitation, UpdateWeights, and Output. Setup stage prepares the data structures necessary for simulation by creating and initializing parameters, matrices, and vectors. CalcExcitation calculates the excitation on each neuron and determines if the excitation is large enough to exceed the threshold for neuron to fire. UpdateWeights updates the synaptic weight for each connection of neurons in each time step. Output outputs the Z matrix to a file for each simulation. Setup and Output only execute once for the duration of the application, and CalcExcitation and UpdateWeights execute for each time step.

The assumption can be easily made that CalcExcitation and UpdateWeights are places to parallelize since they perform the most computations and are called repeatedly. Also, the algorithm is iteratively applied to each neuron, thus the set of neurons is a reasonable choice for the domain that GPU threads concurrently executes on. However, the execution time analysis is still valuable in determining the rather time-consuming parts of the application and understanding the execution time breakdown. Figure 2 shows the structure of the code along with proportions of each step to the total execution. Note that CalcExcitation and UpdateWeights consume more than 99% of the total execution time, which supports the assumption and clarifies the direction of our implementation.

Setup 0.14%

CalcExcitation and UpdateWeights 99.83%

For *Number of Simulations* {
For *Number of Time Steps* {
For *each neuron*
CalcExcitation **42.08%**

For *each neuron*
UpdateWeights **53.55%**

}
}

Output 0.03%

Fig. 2. Execution progresses from top to bottom. The main computation consumes majority of total execution time.

For-loops show the iteration pattern of NeuroJet. The outermost for-loop represents single trial of simulation, and each trial consists of a set number of time steps (the second outermost for-loop). As shown in the figure, the user can run multiple trials of simulation, in which Z matrix must be reinitialized each time. The number of simulations and time steps typically range up to 100 each.

In our implementation, two innermost for-loops are parallelized on GPU, each accountable for a single kernel, and other parts of the application still runs on Octave. In order to run parts of the code on GPU from Octave, we must take advantage of Octave's oct-file interface. Octave can use compiled C++ code as a dynamically linked extension through oct-file interface (comparable to MEX interface in MATLAB). To run CUDA code from Octave, CUDA code must be compiled as a shared object, which can be linked to a C++ code that interfaces between Octave and CUDA [4]. We call this *Octave-CUDA* version of NeuroJet as opposed to the *Octave-Only* version. In fact, we wrote an intermediate version between Octave and Octave-CUDA, called *Octave-C*, for the purpose of performance comparison which follows exactly the same computation steps as in Octave-CUDA, only that CUDA part runs sequentially in C. Later, we also ported the rest of the application onto GPU, having the entire application run on CUDA. This does not mean the application is parallelized from top to bottom: some parts of the application are not parallelizable and they still run in sequential manner. They simply do not rely on Octave anymore. We call this version *CUDA-Only*.

Below are pseudo-codes of calculating the excitation and updating the weights for each version. Note that the readability of the code successively reduces and it becomes harder to understand when written in the actual code. The optimizations applied to Octave-CUDA or CUDA-Only version makes the code further incomprehensible.

Example skeleton code for Octave-Only version.

```
for Number of Neurons{
  Compute excitation for each connection
}
for Number of Neurons{
  Compute new weights for each connection
  Compute new FB and FF weights for each neuron
}
```

Example skeleton code from Octave-C version.

```
for Number of Neurons{
  for Number of Connected Neurons{
    Compute numerator
  }
  for Number of Neurons{
    Compute part of denominator
  }
  Compute excitation
}
for Number of Neurons{
  Compute sum of Z
}
for Number of Neurons{
  for Number of Connected Neurons{
    Compute new weights
  }
  Compute new FB and FF weights
}
```

Example skeleton code from Octave-CUDA or CUDA-Only version

```
calcExcitationKernel{
  for Number of Connected Neurons{
    Compute numerator
  }

  for Number of Thread Blocks{
    Compute part of denominator
    using values from previous kernel
  }
  Compute excitation

  for Number of Thread Blocks{
    Compute partial sum of Z
```

```
  }
}


updateWeightsKernel{
  for Number of Connected Neurons{
    Compute new weights
  }

  for Number of Thread Blocks{
    Compute final sum of Z
  }
  Compute new FB weights and FF weights

  for Number of Thread Blocks{
    Compute partial sum for next kernel
  }
}
```

3.2 Performance

All measurements are made by running the application on the system with NVIDIA Tesla C1060 GPU with 1.3GHz processor clock and Intel Core2Duo CPU with 3.0GHz clock frequency. We performed 60 simulations of 8000 neurons with 35 time steps. Figure 3 shows the graph of each version's performance in terms of execution time. Note that it is in logarithmic scale in order to make the differences interpretable. Performance dramatically improves from Octave-Only to Octave-C version and less dramatically from Octave-C to Octave-CUDA and Octave-CUDA to CUDA-Only version. All of the optimizations discussed in the next subsection are applied in CUDA-Only version. Octave-CUDA version only implements up to GPU Reduction using Shared Memory since Reduced Memory Allocation and Transfers is only applicable when the entire code runs on CUDA.

One of the main reasons for surprisingly poor performance of original Octave-Only version is the lack of JIT compiler support in Octave. The original Octave-Only version seems to have very little optimization done. It extensively uses for-loops when performing matrix-matrix or matrix-vector multiplication. In MATLAB, JIT compiler optimizes these for-loops and can perform in almost native performance. When exactly the same Octave script is run on MATLAB, the performance improves by two orders of magnitude without any changes in the script. Although this may delineate that the original Octave is working inefficiently, the performance on MATLAB is still significantly inferior to other versions involving CUDA. Due to the unrealistically long execution time of original Octave-Only, the performance is extrapolated by executing with smaller iterations.

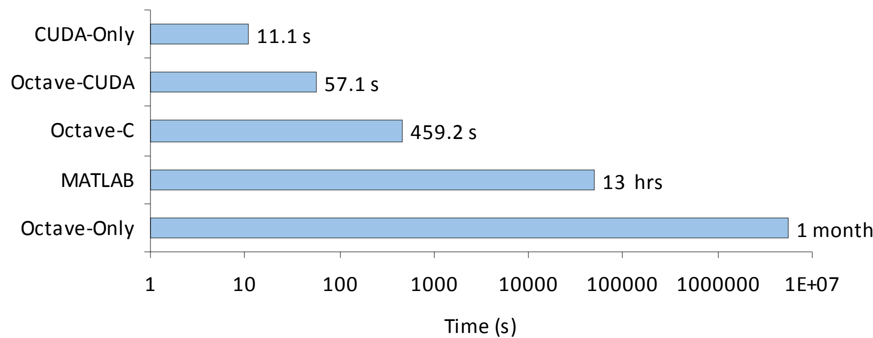

Fig. 3. The x-axis is in logarithmic scale due to too large difference in performance. Note how the performance improves from weeks to hours to minutes to seconds.

An interesting difference exists between CUDA-Only and Octave-CUDA. By moving Setup and Output stages onto GPU, thereby executing the application entirely on GPU, the performance improves by more than 4X. Our profiling on NeuroJet shows that Setup and Output only account for small proportions of the entire application, yet the effect seems to be more significant than what is expected. We show in the next section that this is indeed due to more than just executing on CUDA, but rather an implicit effect of it.

3.3 Optimization

The performance of CUDA-Only version or Octave-CUDA version is a result of a series of optimization techniques. We discuss the GPU-specific optimizations applied to NeuroJet here.

Coalesced Memory Transactions. GPU's global memory transaction is best optimized when threads in a half-warp (a group of 16 threads) access their adjacent locations in memory. For a fully coalesced access, GPU can make a single memory transaction to fulfill the half-warp's load. For a fully uncoalesced access, GPU must make 16 memory transactions for each of the thread's load, thus coalescing memory accesses has significant impact on performance. This implies that traversing arrays in row-major order causes the performance to suffer from uncoalesced accesses. Unlike in CPU, in which accessing arrays in row-major order typically makes an efficient usage of caches, accessing arrays in column-major order, effectively, "SIMD-major" order, is preferable in GPU (Figure 4) [1], [2]. This is in fact done naturally in NeuroJet since

Octave stores matrices in column-major order.

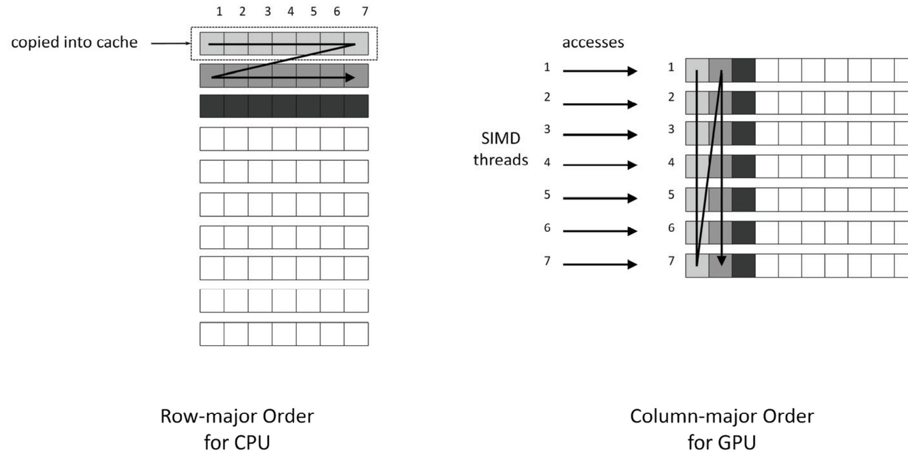

Fig. 4. For CPUs, caches take the most advantage of locality when traversing in row-major order. For GPUs, traversing in column-major order allows SIMD threads to access contiguous blocks of memory.

Utilizing Different Memory Space. In addition to global memory space, GPU provides especially fast per-block shared memory (PBSM), which is shared only by threads in the same thread block. NeuroJet utilizes PBSM by performing a parallel reduction on GPU. Each of CalcExcitation and UpdateWeights involves reduction from the entire array. Since communicating over global memory is too expensive, whereas intra-block communication can be done quickly with PBSM, GPU reduction typically involves 1) one kernel for per-block reduction and another kernel for global reduction, 2) or single kernel for per-block reduction and CPU computes the final reduction. CalcExcitation and UpdateWeights execute as separate kernels in NeuroJet, which means that either each of them must launch an extra kernel to perform final reduction, or that CPU must serially perform the final reduction after each kernel execution. Instead, we take advantage of having two separate kernels: each kernel perform the partial reduction for the other and obtains the result of the partial reduction done by the other kernel.

GPU also provides read-only texture memory and constant memory space. They are cached into texture caches and constant caches respectively, and 3 SMs share a texture and constant cache. Due to an unusual caching property of texture cache, uncoalesced access patterns may obtain better performance by using texture memory space. For that reason, we use texture memory for a randomly accessed matrix. Octave-CUDA and CUDA-Only versions of NeuroJet use GPU reduction technique and texture memory.

CPU-GPU Communication. The overhead of transferring data between CPU and GPU is very expensive and often becomes the bottleneck for the performance.

Programmers typically parallelize functions that are compute-intensive, time-consuming, and repeatedly called, which results in a repeatedly called CPU-to-GPU "preparation" stage. This causes redundant GPU memory allocation and memory transfers between CPU and GPU. Whenever possible, programmers should attempt to apply the for-loop that executes the function iteratively to kernel execution only; if possible, even moving it inside the kernel execution. However, global barrier (separate kernel calls) often prevents the for-loop to move into the kernel. The key is to reduce redundant CPU-GPU communication.

Only the CUDA-Only version of NeuroJet is able to apply this optimization. A part of the setup stage is included in the outermost for-loop, thus second outermost for-loop is the farthest one that can loop over kernel calls in Octave-CUDA version unless the setup stage is implemented in CUDA. The CUDA-Only version moves the outermost for-loop another level in, which reduces redundant CPU-GPU communication. Also, since data structures are created and initialized by GPU and stay on GPU global memory, there is no need for a CPU-to-GPU data transfer.

Figure 5 shows the speedup of each optimized versions of NeuroJet over the performance of original Octave version. Uncoalesced naïve CUDA version is included for comparison purposes in order to see the performance difference between coalesced and non-coalesced global memory access. Notice the significant performance improvement from GPU Reduction to Reduced Redundancy. Moving the setup stage from Octave to CUDA contributes to some performance increase. More so does the effect of reduced GPU memory allocation and data transfers between CPU and GPU, enabled by moving the setup stage to CUDA.

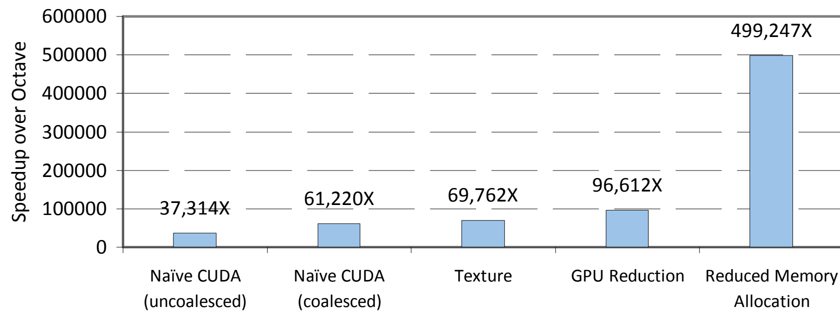

Fig. 5. Speedup of incrementally optimized NeuroJet versions over MATLAB (optimized Octave version).

Certain optimization techniques can be applied independently, e.g. texture memory, whereas others may need to depend on other optimizations. Some other optimizations are not intuitive and not globally applicable, e.g. GPU reduction. Thus, programmers need to examine the data structure, memory access pattern, and iteration patterns to apply

appropriate optimization techniques. Specifically in GPU programming, exploiting GPU's memory bandwidth and memory hierarchy are essential for best performance. Programmers must also consider the overhead of GPU's memory operations and CPU-GPU communication and minimize the cost associated with them.

4 Discussion

4.1 Performance Bottleneck

The neuroscience algorithms used in NeuroJet are not compute-intensive algorithms. Although simple, they require values from many fields. When mapped to the computer simulation, this often means that computation-communication ratio is very low and processor is often waiting for memory transactions to complete. We conducted an analysis on the performance bottleneck of NeuroJet by looking at how it utilizes GPU's memory bandwidth. We also injected extra computations that contribute to the final output of the program. These extra computations are intensive enough to generate a difference in execution time, but not trivial enough to be optimized by compilers. If NeuroJet is bandwidth-bound, adding extra computations should not increase the execution time because processor is spending that computation time waiting for memory to bring the data anyway. If it is compute-bound, then small extra computation adds to the execution time.

Our analysis shows that NeuroJet only utilizes 57% of the peak memory bandwidth of Tesla C1060 GPU (102GB/sec). Of course, most applications do not achieve utilization near the peak, yet NeuroJet only utilizes half of what this GPU can do. This, however, is not sufficient to conclude that memory is waiting for the processor to finish its work and therefore NeuroJet is compute-bound. It only suffices to say NeuroJet is not utilizing the bandwidth enough. Yet, our further experiment with the addition of extra computations results in the same execution time before and after. It indicates that processor is waiting for data from memory, thus the performance of NeuroJet is limited by GPU's global memory bandwidth.

Many applications share similar characteristic of NeuroJet in high ratio of memory access to computation. These applications are expected to exhibit similar behavior. This leads programmers to look for GPU's faster on-chip memory, but there still exist limitations. Per-block shared memory, for instance, has only 16KB of shared space and is not persistent through multiple kernels. Launching multiple kernels therefore require loading into shared memory every time a kernel executes. Aside from programmer's effort on optimizing, reduced memory latency and higher memory bandwidth are essential in next generation GPUs. Some may argue that GPUs are designed to suit better for computationally intensive applications rather than bandwidth-bound or latency-bound

applications. Nonetheless, Rodinia Benchmark Suite [2] clearly shows that many applications are indeed bandwidth-bound and future GPUs should provide resolution to this problem as a general-purpose many-core processor.

4.2 Tradeoffs

GPU and CUDA together provide enormous performance boost to applications that exhibit parallelism. Then, why not write all programs in CUDA if it runs so much faster and saves time? Why not if it runs only a little faster? Higher performance never comes without any price. Tradeoffs associated with performance can be money, energy, space, and so on and so forth. Then, what is the tradeoff of earning performance with CUDA? We use NeuroJet to illustrate that the tradeoff is the loss of simplicity and readability.

Recall the Octave-CUDA version and CUDA-Only version of NeuroJet. The difference between those two versions is the platform of the application. Octave-CUDA runs on Octave and calls outside CUDA code to do the heavy work on GPU, while CUDA-Only version runs entirely on CUDA. Algorithms and data structures are identical in both versions, yet the performance differs by 4X. Setting up input data requires numerous operations on matrices and arrays and Octave provides an incredibly convenient set of commands to manipulate matrices that makes the setup stage easy to read, modify, and understand. This on CUDA takes much more heavy operations such as memory allocation, memory transfers, and kernel calls, yet the code is not quite easy to understand without closely examining it. We acknowledge 4X speedup as a significant performance difference. However, for users who want both the readability and performance, porting parts of code to CUDA and leaving remaining parts to original platform may be more desirable. Writing a code for the setup stage in CUDA takes far more lines of code and complexity, where it can be done in only a few lines in Octave. For CPU programs, basic libraries exist to allow programmers to take advantage of modularity. In GPU, these library calls are generally separate calls and the overhead is significant. We believe the next generation GPU or GPU programming model must address this problem of expensive modularity between CUDA and other languages through compiler support.

5 Conclusion

The parallel property of modern multi- and many-core processors does not guarantee scalable performances in all applications. Applications need to be specifically programmed for multi- and many-core processors. We have demonstrated with NeuroJet, that it requires thorough understanding of the application’s property, particularly the data structure and memory access pattern. Programmers must have a comprehensive

understanding of the target platform as well. The results of our highly parallel implementation of NeuroJet on GPU show the promising performance improvement of up to 500,000X speedup, though deciding whether such speedup is worth trading with readability of the code depend on purpose of whom using the code. Our techniques used for profiling, implementation, optimization, and analysis are meant to be used by other researchers to study their applications if not yet so. Some of the insights discovered throughout the process of implementation and analysis of NeuroJet are not very intuitive. They also are to help researchers with their work anyway applicable. Finally, they are to contribute to the development and enhancement of new and current multi- and many-core processors as well as corresponding programming models.

References

1. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A Performance Study of General Purpose Applications on Graphics Processors using CUDA. Journal of Parallel and Distributed Computing, 68(10), 1370-1380 (2008)
2. Che, S., Boyer, M., Meng, J., Tarjan, D., Lee, S., Sheaffer, J.W., Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: IEEE International Symposium on Workload Characterization (IISWC), 44-54 (2009)
3. Chaves, J.C., Nehrbass, J., Gulifoos, B., Gardiner, J., Ahalt, S., Krishnamurthy, A.: Octave and Python: High-Level Scripting Language Productivity and Performance Evaluation. In: HPCMP User Group Conference (2006)
4. Goh Cheng Teng Advanced Computing Group.: Matrix Multiplication on GPU in Octave. Technical report, Institute of High Performance Computing (2008)
5. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable Parallel Programming with CUDA. ACM Queue, 6(2), 40-53 (2008)
6. Kanter, D.: NVIDIA's GT200: Inside a Parallel Processor. Real World Technologies, http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=6
7. NVIDIA Corporation.: NVIDIA Tesla C1060 Computing Processor. NVIDIA, http://www.nvidia.com/object/product_tesla_c1060_us.html
8. NeuroJet, http://www.neurojet.com
9. Eaton, J.W.: GNU Octave Manual. Network Theory Limited (2002)
10. Howe, A., Levy, W.B.: A Hippocampal Model Predicts a Fluctuating Phase Transition when Learning Certain Trace Conditioning Paradigms. Cognitive Neurodynamics (2007)