TEACHING DESIGN & ANALYSIS OF MULTI-CORE PARALLEL ALGORITHMS USING CUDA*

Quoc-Nam Tran, Lamar University

ABSTRACT

One of the dominant trends in microprocessor architecture in recent years is continually increasing chip-level parallelism. However, many undergraduate curriculums, especially at small schools, do not offer courses that focus on the design and analysis of multi-threaded algorithms for multi-core processors. The courses that are offered address the theoretical aspects of parallel system design, but often fail to provide students with the opportunity to develop and evaluate distributed applications in real-world environments. As a result, undergraduate students are not as prepared as they should be for graduate study or careers in industry.

We fill these gaps by creating multi-threaded programming (MTP) materials which target multi-core processors for the junior-level in computer science. The MTP materials are used in the second half of the Design and Analysis of Algorithms course replacing the theoretical aspects of parallel system design using the PRAM model. In order to overcome the difficulty of teaching and learning multi-core programming, problem-based learning (PBL) is used to ease students' transition from the single-threaded programming model to MTP. The course is well-suited for both large and small schools, and will greatly contribute to CS education and preparation of CS students for graduate study or careers in industry.

1 Introduction

Facing increasing power dissipation and little instruction-level parallelism left to exploit, computer architects are realizing further performance gains by placing multiple "slower" processor cores on a chip rather than by building faster uni-processors. One of the dominant trends in microprocessor architecture in recent years has been continually increasing chip-level parallelism. Multi-core CPUs with 2–4 cores are now commonplace

---

and there is every indication that the trend towards increasing parallelism will continue on towards "manycore" chips that provide far higher degrees of parallelism. In fact, new massive parallel computing processors such as the Nvidia's Testla-C1060 already have 240 cores. As a consequence, application performance will increasingly depend on the use of parallelism. On multiprocessors, this is often achieved through the use of threads.

However, many undergraduate curriculums, especially at small schools, do not offer courses that focus on the design and analysis of multi-threaded algorithms for multi-core processors. The courses that are offered address the theoretical aspects of parallel system design, but often fail to provide students with the opportunity to develop and evaluate distributed applications in real-world environments. As a result, undergraduate students are not as prepared as they should be for graduate study or careers in industry.

We propose to fill the aforementioned gaps by creating multi-threaded programming (MTP) materials targeting multi-core processors for the junior-level and graduate-level students in computer science. The MTP materials will be used in the second half of the Design and Analysis of Algorithms course replacing the theoretical aspects of parallel system design using PRAM model we currently use. In order to overcome the difficulty of teaching and learning multi-core programming, problem-based learning (PBL) is used to ease students' transition from the single-threaded programming model to MTP.

We develop our MTP materials using the CUDA multi-threaded programming environment from Nvidia to help reducing the learning curve for the students since CUDA is a C-like programming language with minimal training on additional features for controlling multi threading. CUDA also has wrappers available for Python, Fortran and Java. Furthermore, we use Nvidia's computing processors (GPU) with 216 cores and 1GB of RAM for our PBL course materials because these computing processors can offer large performance benefits (a speed-up of 100X for many applications in comparison with a single core program) to motivate the students.

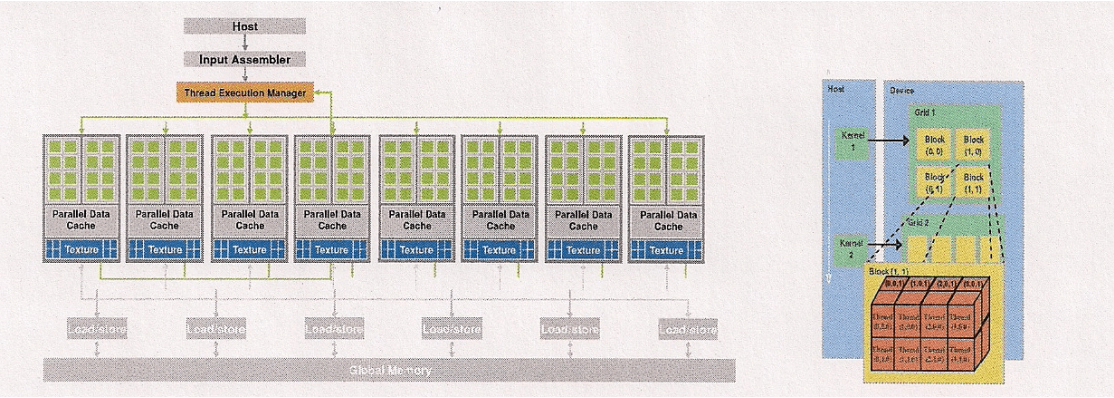


Figure 1: CUDA hardware interface & programming model

It is generally acknowledged that developing correct multi-threaded codes is difficult, because threaded may interact with each other in unpredictable ways. The newly structured courses with a variety of problem-based and lecture-based methods of instruction will:

1. Help students to learn principals of designing, implementing and analyzing multi-threaded algorithms for multi-core computing processors.

2. Develop students with theoretical computer science backgrounds and the mathematical and analytical maturity necessary to allow them to adapt to changing requirements in the job market or enter graduate programs in computer science.
3. Instill within students the technical, communicative, and character skills that will prepare them for future roles.

The course leverages the use of CUDA, multi-core computing processors and open-source software packages to allow students to develop large-scale systems using a minimal amount of local computing resources, making the course well-suited for both large and small schools.

The paper is organized as follows: In Section 2, the author explain why CUDA and multi-core computing processors offer new opportunities for educators to give students access to distributed resources, even when budgetary constraints are present. In Section 3, the author presents some PBL learning materials targeting multi-core processors.

2 CUDA Model and Significance

2.1 Related Work

Throughout the past two decades, several educators have proposed ways to introduce students to distributed computing concepts [1]. In the 1990s, the key challenges were developing courses that emphasized "practical case studies" without requiring expensive equipment and high performance machines [2 ,3 ,17]. In recent years, the need for expensive supercomputers has diminished, and the emphasis of many distributed computing courses now focuses on realistically simulating and emulating large-scale distributed systems using local resources [7 ,9 ,15]. The aforementioned previous work had two ideas in common: (i) a general dissatisfaction with the lack of undergraduate distributed computing courses; and (ii) an acknowledgment of the importance of integrating theory and practice to give students exposure to realistic systems. While the state-of-the-art of distributed computing has changed significantly over the past few decades, these two ideas are still relevant today.

The emergence of massive parallel computing platforms such as Nvidia Testla-C1060 has created new opportunities for educators to give students access to distributed resources, even when budgetary constraints are present.

2.2 CUDA Models

Commodity graphics processors (GPUs) have been at the leading edge of the multi-core drive towards increased chip level parallelism for some time and are already fundamentally manycore processors. Current Nvidia GPUs, for example, contain up to 240 scalar processing elements per chip [11], and in contrast to earlier generations of GPUs, they can be programmed directly in C using CUDA [13 ,14].

GPUs have become a cost-effective parallel platform to solve many general problems. Many problems in the fields of linear algebra, image processing, computer vision, signal processing, etc., have benefited from its speed and parallel processing capability [8]. GPU clusters have also been used to perform compute intensive tasks, like

finite element computations, gas dispersion simulation, heat shimmering simulation, accurate nuclear explosion simulations, etc. [19 ,4]

On an abstract level, all the 240 processors of the Nvidia Testla-C1060 are of same type with similar memory access speeds, which makes it a massive parallel processor. CUDA is a programming interface to use this parallel architecture for general purpose computing. This interface is a set of library functions which can be coded as an extension of the C language. A compiler generates executable code for the CUDA device. The CPU sees a CUDA device as a multi-core co-processor. The CUDA design does not have memory restrictions of GPGPU. One can access all memory available on the device using CUDA with no restriction on its representation though the access times vary for different types of memory. This enhancement in the memory model allows programmers to better exploit the parallel power of the Testla-C1060 processor for general purpose computing.

2.2.1 CUDA Hardware Model

At the hardware level, the Testla-C1060 processor is a collection of 30 multiprocessors, with 8 processors each. Each multiprocessor has its own shared memory which is common to all the 8 processors inside it. It also has a set of 32-bit registers, texture, and constant memory caches (Figure 1). At any given cycle, each processor in the multiprocessor executes the same instruction on different data, which makes each a SIMD processor. Communication between multiprocessors is through the device memory, which is available to all the processors of the multiprocessors.

2.2.2 CUDA Programming Model

For programmers, the CUDA model is a collection of threads running in parallel. A warp is a collection of threads that can run simultaneously on a multiprocessor. The warp size is fixed for a specific GPU. The programmer decides the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of threads (called a block) runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared. A single execution on a device generates a number of blocks. A collection of all blocks in a single execution is called a grid (Figure 1). All threads of all blocks executing on a single multiprocessor divide its resources equally amongst themselves. Each thread and block is given a unique ID that can be accessed within the thread during its execution. Each thread executes a single instruction set called the kernel. The kernel is the core code to be executed on each thread. Using the thread and block IDs each thread can perform the kernel task on different set of data. Since the device memory is available to all the threads, it can access any memory location. The CUDA programming interface provides an almost Parallel Random Access Machine (PRAM) architecture, if one uses the device memory alone. However, the multiprocessors follow a SIMD model, the performance improves with the use of shared memory which can be accessed faster than the device memory. The hardware architecture allows multiple instruction sets to be executed on different multiprocessors. The current CUDA programming model, however, cannot assign different kernels to different multiprocessors, though this may be simulated using conditionals. With CUDA, the GPU can be viewed as a massive parallel SIMD processor, limited only by the amount of

memory available on the graphics hardware. The Testla-C1060 card has 4GB memory. Large graphs can reside in this memory, given a suitable representation. The problem needs to be partitioned appropriately into multiple grids for handling even larger graphs.

3 PARALLEL DESIGN & ANALYSIS

In this section, we present some learning materials and strategies we developed and used for the Design and Analysis of Algorithms course in Spring and Fall 2009. As students have learned graph algorithms in the first half of the course, we create some PBL materials for multi-threaded graph algorithms. Graph representations are common in many problem domains including scientific and engineering applications. Fundamental graph operations like breadth-first search, depth-first search, shortest path, etc., occur frequently in these domains. Some problems map to very large graphs, often involving millions of vertices. For example, problems like VLSI chip layout, formal verification, data mining, and network analysis can require graphs with millions of vertices. While fast implementations of sequential fundamental graph algorithms exist they are of the order of number of vertices and edges. Such algorithms become impractical on very large graphs.

Even though in this section we only present the design and analysis of some fundamental parallel graph algorithms, we have developed and collected materials for many problems in the fields of linear algebra, image processing, computer vision, signal processing, finite element computations, gas dispersion simulation, heat shimmering simulation, accurate nuclear explosion simulations, etc. [4 ,6 ,8 ,19]. We also used information from multiple sources [12 ,14 ,16] to cover: (a) Basic concepts and data types for the CUDA programming model; (b) Basics of CUDA application programming interface; (c) Simple examples to illustrate basic concepts and functionalities; and (d) Compiling and Debugging.

3.1 All Pairs Shortest Path (APSP)

To design a CUDA parallel algorithm for the APSP problem, we use the representation of graphs in an adjacency matrix form. Given a graph *G(V,E)*, the adjacency matrix A[*i,j*] is defined as $w_{i,j}$ if $v_i, v_j \in E$, 0 if $i=j$, and ∞ if $v_i, v_j \notin E$.

```
FloydAPSP( A, n)
  D^0 ← A
  for k from 1 to n
    for i from 1 to n
      for j from 1 to n
        d^k_{i,j} ← min(d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{k,j})
  return D^n
```

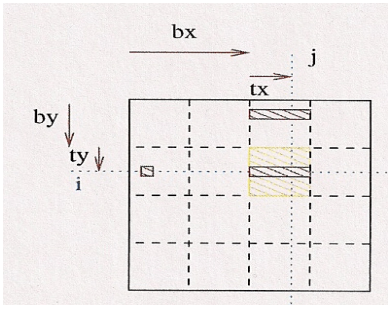
We first design a simple parallel algorithm for APSP using a dynamic-programming formulation [10 ,18]. Assume that $V=\{v_1,v_2,...,v_n\}$. For any pair of vertices $v_i, v_j \in V$, consider all paths from v_i to v_j whose intermediate vertices are drawn from a subset $\{v_1,v_2,...,v_k\}$ for some k. Let p be a minimum-weight simple path, the Floyd-Warshall algorithm is based on the relationship between path p and shortest paths from v_i to v_j with all intermediate vertices in the set $\{v_1,v_2,...,v_{k-1}\}$. The relationship depends on whether or not v_k is an intermediate vertex of path p.

- If v_k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{v_1,v_2,...,v_{k-1}\}$. Thus, a shortest path from vertex v_i to vertex v_j with all all intermediate vertices in the set $\{v_1,v_2,...,v_{k-1}\}$ is also a shortest path from vertex v_i to vertex v_j with all all intermediate vertices in the set $\{v_1,v_2,...,v_k\}$.
- If v_k is an intermediate vertex of path p, then we can break p down to $v_i \xrightarrow{p_1} v_k \xrightarrow{p_2} v_j$. Since any sub-path of a shortest path is also shortest, p_1 is a shortest path from v_i to v_k with all intermediate vertices in $\{v_1,v_2,...,v_{k-1}\}$. Similarly, p_2 is a shortest path from v_k to v_j with all intermediate vertices in $\{v_1,v_2,...,v_{k-1}\}$.

Based on the above observations, an n^3 sequential algorithm has been discussed in [5].

3.2 Parallel Design

A parallel design for the Floyd algorithm would be straight forward by letting one thread to work on one element of the matrix $D^k_{i,j}$ for k from 1 to n. To distribute the computation to the calculating cores, we partition the matrix D^k into tiles with size 16 × 16. The intention is that the threads for calculating the elements within a tile will be fit into a block of threads for one streaming multi-processor (SM) in Cuda. Each of the SM in a GPU can only handle at most 512/1024 co-resident concurrent threads. A CUDA kernel is executed by an array of threads. Threads are executed in groups of 32 called warps. All threads run the same code (SPMD). It is worth noting that CUDA threads are of lighter weight and take very few cycles to generate and schedule due to efficient hardware support. Each thread has an ID that it uses to compute memory addresses and make control decisions. In the figure, bx and by are the coordinates of the tile, which can be identified by using Cuda's blockIdx. Similarly, tx and ty are the coordinates of the element within the tile, which can be identified by using Cuda's threadIdx. The function FloydKernel is run simultaneously on all cores.

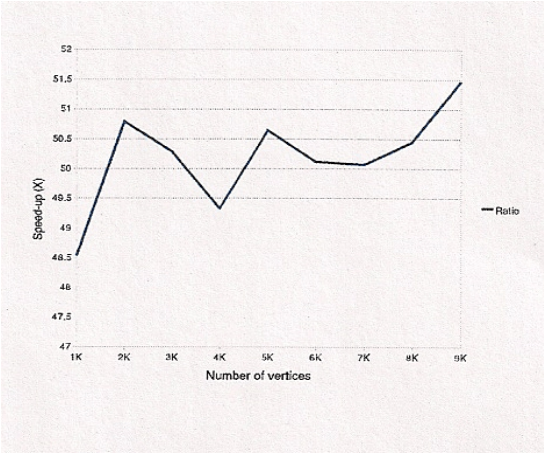



```
CudaFloyd (A, P, n)
  Allocate Ad ← A and Pd on the device
 for k from 1 to n
   FloydKernel(Ad, Pd, k)
   Ad ← Pd on the device
 P ← Pd
```

```
FloydKernel (A, P, k)
  Identify the location to work on;
  BLOCK_SIZE = 16;
 i = by * BLOCK_SIZE + ty;
    j = bx * BLOCK_SIZE + tx;
    P_{i,j} ← min(A_{i,j}, A_{i,k} + A_{k,j})
```

3.3 Performance & Analysis

Algorithm CudaFloyd runs in $\Theta(n^3/|P|$ where $|P|$ is the number of cores. However, with all calculating cores sharing the matrices *Ad* and *Pd* on the global memory, the performance gains for the parallel CudaFloyd algorithm will not be high. We experimented our implementations using Cuda 2.3 under Linux Ubuntu 8.10 on a Dell Dimension E521, Athlon 64 3200+ with 4GB, a GeForce GTX-260 GPU with 216 cores and 896MB GDDR3. We report our experiments with dense graphs with 1K-10K vertices. The results and running times of the serial and parallel implementations were compared. It shows a performance speed-up of 48x-52x for the parallel algorithm. It is worth to notice that in our implementation we utilize the GPU's the shared memory with a high memory bandwidth of 111.9GB/sec and coalesce the use of the global memory.



4 CONCLUSION

We presented the design and analysis for some of our multi-threaded algorithms for multi-core processors. The multi-threaded programming materials are used in the second half of the Design and Analysis of Algorithms course replacing the theoretical aspects of parallel system design using PRAM model. We leverage the use of CUDA, multi-core computing processors and open-source software packages to allow students to develop large-scale systems using a minimal amount of local computing resources, making the course well-suited for both large and small schools.

REFERENCES

[1] Albrecht, J. R. Bringing big systems to small schools: Distributed systems for undergraduates. In Proceedings of SIGCSE'09 (2009), pp. 101–105.

[2] Cunha, J. C., and Lourenco, J. An integrated course on parallel and distributed processing. In Proceedings of SIGCSE'98 (1998).

[3] Dillon, E., Santos, C. G. D., and Guyard, J. Teaching an engineering approach for network computing. In Proceedings of SIGCSE'97 (1997).

[4] Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. Gpu cluster for high performance computing. In Proceedings of the 2004 ACM/IEEE conference on Supercomputing SC'04 (2004), IEEE.

[5] Floyd, R. Algorithm 97 (shortest path). Communications of the ACM 5, 6 (1962), 345.

[6] Harish, P., and Narayanan, P. J. Accelerating large graph algorithms on the gpu using cuda. Tech. rep., International Institute of Information Technology Hyderabad, India, 2008.

[7] Hnatyshin, V. Y., and Lobo, A. F. Undergraduate data communications and networking projects using opnet and wireshark software. In Proceedings of SIGCSE'08 (2008).

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

[9] Laverell, W. D., Fei, Z., and Griffioen, J. N. Isn't it time you had an emulab?

[10] Lawler, E. Combinatorial Optimization: Networks and Matroids. Holt, Rinehart, and Winston, 1976.

[11] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. Nvidia tesla: A unified graphics and computing architecture. IEEE Micro 28, 2 (2008).

[12] mei Hwu, W., and Kirk, D. http://courses.ece.uiuc.edu/ece498/al, 2008.

[13] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda. Queue 6, 2, 40–53.

[14] NVIDIA Corporation. NVIDIA CUDA Programming Guide, June 2008. version 2.0.

[15] Pheatt, C. An easy to use distributed computing framework. In Proceedings of SIGCSE'07 (2007).

[16] Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore gpus. In Proc. 23rd IEEE International Parallel and Distributed Processing Symposium (2009).

[17] Stewart, C. Distributed systems in the undergraduate curriculum. ACM SIGCSE Bulletin 26, 4 (1994).

[18] Warshall, S. A theorem on boolean matrices. Journal of the ACM 9, 1 (1962), 11–12.

[19] Zhao, Y., Han, Y., Fan, Z., Qiu, F., Kuo, Y.-C., Kaufman, A. E., and Mueller, K. Visual simulation of heat shimmering and mirage. IEEE Transactions on Visualization and Computer Graphics 13, 1 (2007), 179–189.