Temperature Dispersion: Many-Core vs. Traditional Multi-Core Laplace Transform Implementation

A. Knowles[1], **E. Colmenares**[2]
[1,2]Department of Computer Science, Midwestern State University, Wichita Falls, Texas, USA

Abstract—*As a common scientific kernel, the heat equation arises in the modeling of several natural phenomena such as weather simulations. Over the last decade, computational parallel algorithms written for many core (GP-GPU) architectures have provided more efficient performance than their multi-core predecessors. Along with being more computationally efficient for tasks that exhibit data parallelism, GP-GPU supercomputers are also more energy efficient. Thus, it could be worthwhile in the long run to port current multi-core based weather simulations to a GP-GPU environment. In this paper, a parallel Laplace implementation in Cuda was compared against a parallel Laplace implementation in MPI. Timings were acquired for both communication and computation separately for the many-core parallel version. This paper concludes with a discussion of efficiency comparisons between the many core and multi-core implementations, as well as a discussion of resources required to port a multi-core implementation to a many core architecture.*

Keywords: parallel, asynchronous algorithms, Jacobi, green computing

1. Introduction

This paper investigates an asynchronous version of the heat equation algorithm developed in Cuda, which determines a solution to sets of linear equations $Ax = b$, where A is a large $n \times n$ matrix, and x and b are vectors. This algorithm, in its basic form, lies at the heart of numerous important scientific kernals making efficient implementations crucial. Existing implementations of this algorithm tend to be synchronous and developed for multi-core architectures using MPI. These synchronous implementations require that the vector information be updated and communicated among the participating network, halting computation at the cost of communication. This creates a barrier beyond which computation cannot proceed until the vectors have been updated for each iteration, meaning that these implementations will be difficult, if not impossible, to scale to millions of cores[4].

Here, we focus on implementing an asynchronous algorithm that avoids this blocking behavior and allows processors to perform calculations with the data that they have access to, regardless of whether new data has arrived. There has been work on the theoretical (Chazan and Miranker, 1969; Baudet, 1978; and Bertsekas and Tsitsiklis, 1989) and on practical implementations (Bull and Freeman, 1992; Bahi et al., 2006, de Jager and Bradley, 2010). In this paper, we will look at the performance of this algorithm experimentally by adapting the Jacobi method, the simplest iterative algorithm, for a many core architecture. In order to reach temperature convergence for Jacobi iterations of the Laplace formula, numerous calculations must be performed each Jacobi iteration. However, note that all of these calculations use the exact same formula, only the data changes, making this algorithm perfectly suited to a many-core architecture. General Purpose GPU (GP-GPU) architectures, also referred to as many-core architectures, shine when an algorithm can be adapted to the Single Instruction, Multiple Data (SIMD) paradigm[10]. This is because the architecture allows for simultaneous computation across thousands of cores. The remainder of this paper will discuss methodologies used to adapt the Laplace algorithm to a many-core architecture using Cuda C, results from the adaptation, as well as a short discussion on the energy efficiency of many-core architectures and costs associated with porting algorithms designed for a multicore system to a many-core system.

2. Laplace algorithm

The basic heat equation has a fundamental importance in many fields. In the field of mathematics, it stands as the basic definition of a partial differential equation, probability theory sees this equation connected with the study of Brownian motion[7], and it has been used to solve the Black-Scholes partial differential equation used in the financial industry[12]. The heat equation can be generally defined in any coordinate system by the next expression:

$$\frac{\partial u}{\partial t} - \alpha \bigtriangledown^2 u = 0 \tag{1}$$

In this simplified form, α represents a positive constant which defines the rate of heat transfer from the hot side to the cold side, $\bigtriangledown^2$ defines the Laplace operator, which is used to determine when the temperature converges and stops changing, and u is a function describing the temperature at a given location.

Jacobi's method of solving a system of linear equations where $Ax = b$, where A is a large $n \times n$ matrix, computes

a sequence of vectors $x^{(k)}$ where,

$$x_i^{(k)} = \frac{1}{a^{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)}\right), i = 1 : n \qquad (2)$$

The $x_i^{(k)}$, and $i = 1 : n$ terms, are both independent, which means that they can be calculated simultaneously in parallel.

Simplifying the Jacobi method to work within the confines of the heat equation, we can calculate temperature at a given point by averaging the temperatures of the points around it, as defined by the following generalized equation:

$$T_{i,j} = 0.25 \times [T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1}] \qquad (3)$$

The Laplace operator then uses this temperature function, updating a matrix of temperatures with each Jacobi iteration until reaching a conversion point.

In order to visualize this problem, let us place our matrix with i columns and j rows across a metal plate as seen in Figure 1. The temperature of a single location on the grid, denoted as point $h_{i,j}$, can be calculated by taking an average of the points directly above ($h_{i-1,j}$), below ($h_{i+1,j}$), to the right ($h_{i,j+1}$), and to the left ($h_{i,j-1}$) of the origin point.

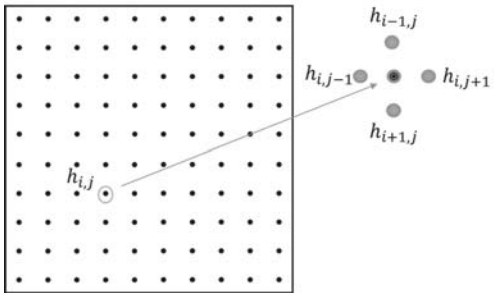

Fig. 1: Illustration of a metal plate divided into a matrix with i columns and j rows An expansion of the points surrounding $h_{i,j}$ is also shown.

The simplicity of the function belies the shear number of calculations required in order to reach a temperature convergence. For instance, map the temperature grid to a small 128×128 matrix and each iteration calculates $16,384$ different temperatures. If we were to grow that matrix to 4096×4096, approximately 16 million calculations would be performed each iteration. Figure 2 shows how rapidly the number of calculations per iteration grows as the size of the matrix grows.

If the right edge of the metal plate were heated using a linear temperature increase, around 2100 iterations would be required to reach a temperature convergence for the 128×128 matrix, meaning over 34 million calculations were required to reach convergence on a small matrix. Increasing the matrix size to 4096×4096 requires just over 3500 iterations to reach convergence, that is more than 58.7 billion calculations.

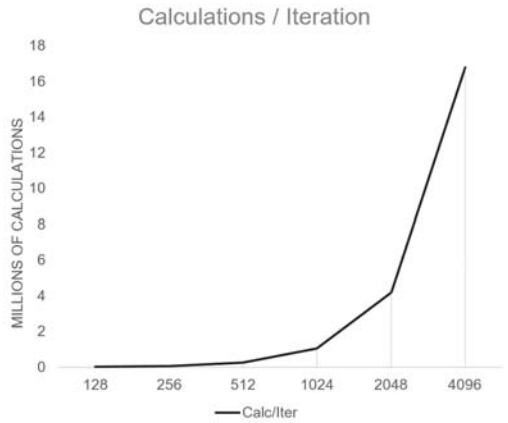

Fig. 2: Graph showing exponential growth rate of the number of temperature calculations required per iteration as matrix size grows.

3. Implementation

To investigate the performance of the Jacobi method algorithm, we have implemented one serial version and one asynchronous parallel version written for a many-core architecture in Cuda using only global memory. We compare our asynchronous parallel many-core version both to the serial version and to an asynchronous MPI version developed by Bethune et al[4]. In the asynchronous version we ensure that a process reads only from the most recently received complete set of data from another process. We chose to solve the 2D diffusion problem where $\bigtriangledown^2 u = 0$ using a 6-point stencil over a 2D grid as shown in Figure 1. We heated the simulated metal plate, represented by an $(n \times n)$ matrix, along the right edge in a linear fashion based on the following formula:

$$T[right] = (100.0/ROW) * i \qquad (4)$$

where ROW is the number of rows in the $n \times n$ matrix, and i is the current counter in the `for` loop used to populate the matrix. We tested 6 matrix sizes: (128×128), (256×256), (512×512), $(1024\ x \times 1024)$, (2048×2048), and (4096×4096), performing 2000 Jacobi iterations on each matrix for each of 5 runs. For the parallel version, we ran each matrix using a block size of 256 threads, and 1024 threads. Grid size grew based on the size of the problem matrix. The following pseudocode defines our computational kernel:

```
_global_ void laplace_on_device
        (float * A, float * B)
  // calculate row and column indexes
  // of the center point handled by
  // this thread
  T[i,j] = T[row, col]
```

```
// retrieve index values for
// center point and neighboring
// points within 1D array
int O = i,j
int U = i, j+1
int D = i, j-1
int R = i-1, j
int L = i+1, j

// update only interior
// node points
if(row < col-1 && col < row-1)
  A[O] = 0.25 *
    (B[U] + B[D] + B[L] + B[R]);
```

First, we map our current thread to a location within the matrix, then we determine the location within the 1D array of the origin point and each surrounding point. If the origin point represented by this thread is an interior node, then the average of the temperature of the surrounding points, stored in the array B, is calculated and stored in array A.

4. Performance Analysis

To investigate the performance of our implementation of the Jacobi method, Turing, a Powerwulf ZXR1+ Cluster HPC Computer Engine, was used. Turing is an academic teaching and research oriented HPC cluster, which provides Midwestern State University with cutting edge HPC technology. The HPC cluster contains 1 head node containing 16 2.1 GHz cores, 9 compute nodes containing 160 2.1GHz cores, and 1 GPU compute node with four Tesla K80s with 9,984 Nvidia Cuda cores among them. There is a 704 GB High Performance ECC System Memory with 4GB memory per compute node processor core and 4GB memory per head node processor core. Turing has 16+ TB of RAW storage space made up of 2 1TB SATA III drives on the head node, 1 14TB SATA III Raw Space on the head node, and 128 GB Solid State SATA III (SSD) Scratch per compute node, as well as 10/100/1000/10000 10GigE High Speed Network Backplane for message passing. The 4 Nvidia Tesla K80s each deliver 8.74 teraflops of single precision performance. Each K80 chip is broken up into two graphics processors, with a combined 4,992 processing cores and 24GB of GDDR5 memory. GPUs are designed with a massively parallel architecture consisting of thousands of tiny, efficient cores making them perfect for handling multiple tasks simultaneously. The massively parallel architecture of GPUs is perfectly suited to run Single Instruction, Multiple Data (SIMD) problems, such as this one, since the same instruction can be performed on a large number of cores simultaneously.

A sample of averaged runtime results for the serial and parallel implementation can be seen in Table 1.

Due to the massively parallel nature of a GP-GPU environment, a massive reduction was seen in the time required to perform 2000 Jacobi iterations, despite the use of global memory. Slightly better performance was also seen in the 1024 thread version over the 256 thread version, as can be seen in Figure 3.

Table 1: Runtime (in seconds)

Version	128	256	512	1024	2048	4096
Serial	0.27	2.52	10.81	49.35	242.35	3602.51
256 Threads	0.07	0.09	0.19	0.55	1.69	5.81
1024 Threads	0.09	0.09	0.17	0.49	1.49	4.92

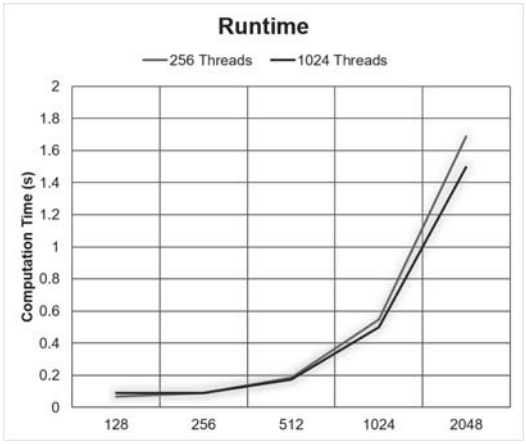

Fig. 3: Graph showing runtime performance difference between the 256 and 1024 thread versions.

In order to determine improvement of the parallel version over the sequential version, the concept of speedup - which allows us to answer the question of exactly how much faster the many-core system can solve the problem under consideration - was used. Speedup is defined as

$$S(n,p) = \frac{T_{serial}(n)}{T_{parallel}(n,p)} \tag{5}$$

where T_{serial} is the runtime for the serial algorithm, $T_{parallel}$ is the runtime for the parallel algorithm, n is the problem size, and p is the number of nodes on which the given problem size has been distributed. When applying this formula to the Cuda environment, p refers to the number of threads used to solve the given problem. The number of threads used for each problem size can be determined by calculating the number of blocks in the grid times the number of threads in a block. As previously stated, our grid size changed depending on the size of the $n \times n$ matrix, but only two different block sizes were used - 256 threads and 1024 threads. Overall thread count for the 256 thread block size ranged from just over $16,000$ threads for the (128×128) matrix to over 16 million threads for the (4096×4096) matrix. Overall thread count for the 1024 thread block size ranged from just over $65,000$ threads for the (128×128) matrix to over 67 million threads for the (4096×4096) matrix. Despite the

use of global memory, we still saw massive speedup over the serial version, as seen in Figure 4.

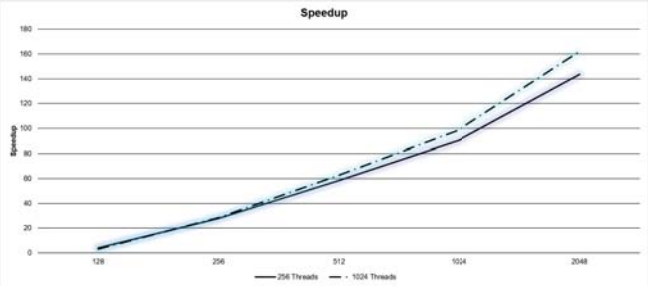

Fig. 4: Graph showing speedup of the two different thread block sizes over the serial version.

Although the performance between our many-core Cuda implementation and the MPI programming model developed by Bethune et al. cannot be directly compared since the problem being solved is slightly different, there was an attempt to compare the time spent on each iteration against the number of processes. As you can see in Figure 5, the traditional multi-core MPI asynchronous implementation performs significantly slower than the asynchronous many-core implementation per iteration.

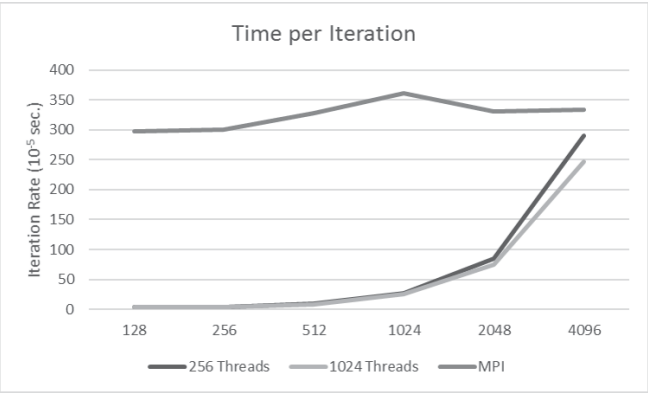

Fig. 5: Comparison of time spent on each iteration based on problem size.

5. Conclusions

MPI is the most widely used programming model on HPC systems, and is the de facto standard for any distributed memory application[5]. However, in this paper we presented a many-core alternative to a widely utilized computational kernel and showed that massive speedup can be achieved with a Cuda implementation over a traditional MPI implementation.

Also, with more and more applications requiring high performance systems that are extremely expensive to operate, it is our responsibility to attempt to make those systems as green as possible. It is well known that in terms of energy costs, GP-GPU computers are more cost efficient than their traditional multi-core counterparts. With greener computing in mind, it would be beneficial to adapt widely used computational kernels to a many-core architecture. Costs associated with porting code from one architecture to another, would, in the beginning be significant. However, it is our belief that with a small learning curve, numerous widely used computational kernels could be adapted to Cuda and see not only improved performance over the MPI version, but also be performed on a greener computing platform, reducing our overall environmental impact.

6. Further Research

It would be interesting to see if a grid size could be found to achieve maximum throughput for each problem size, as opposed to growing the grid size as the problem grew as we did here. We used a global memory scheme in this implementation, but would like to see a shared memory scheme implemented in the future. We would also like to grow the matrix size beyond the capacity of an individual GPU node and implement an MPI/Cuda parallel version.

References

[1] J.M. Bahi, S. Contassot-Vivier, and R. Couturier, *Performance comparison of parallel programming environments for implementing aiac algorithms*, Journal of Supercomputing, 35(3): 226 - 244, 2006.

[2] G. Baudet, *Asynchronous iterative methods for multiprocessors*, Journal of the Association for Computing Machinery, 25(2): 226-244, 1978.

[3] D. Bertsekas, J. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Englewood Cliffs, NJ:Prentice Hall, 1989.

[4] I. Bethune, J.M Bull, N.J. Dingle, and N.J. Higham, *Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP*, The International Journal of High Performance Computing Applications (Sage Publications Sage UK: London, England), 28, 1, 97 - 111, 2014.

[5] A.R Brodtkorb, M.L Saetra, and M. Altinakar, *Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation*, Computers & Fluids, 55, pp.1-12, 2012.

[6] J. Bull, T. Freeman, *Numerical Performance of an asynchronous Jacobi iteration*, Proceedings of the second joint international conference on vector and parallel processing (CONPAR 1992), 361-366, 1992.

[7] J. R. Cannon, *The one-dimensional heat equation*, Cambridge University Press, 1984, vol. 23.

[8] D. Chazan, W. Miranker, *Chaotic Relaxation*, Linear Algebra and It's applications, 2: 199 - 222, 1969.

[9] D. deJager, J. Bradley, *Extracting state-based performance metrics using asynchronous iterative techniques*, Performance Evaluation, 67(12): 1353-1372, 2010.

[10] W.M. Hwu and D. Kirk, *Programming massively parallel processors*, Morgan Kaufmann Publishers, 2009.

[11] S.A. Khuri, *A Laplace decomposition algorithm applied to a class of nonlinear differential equations. Journal of Applied Mathematics*, 1(4), pp.141-155, 2001

[12] P. Wilmott, S. Howison, and J. Dewynne, *The mathematics of financial derivatives: a student introduction*, Cambridge University Press, 1995.