A proposal for a GPU scheduler to optimize large matrix multiplication performance

Kenan Rahmani, CS 497
Department of Computer Science, Purdue University
krahmani@purdue.edu

Advisor: Dr. Zhiyuan Li
Also: Mohammad Z. Siddiqui

Abstract

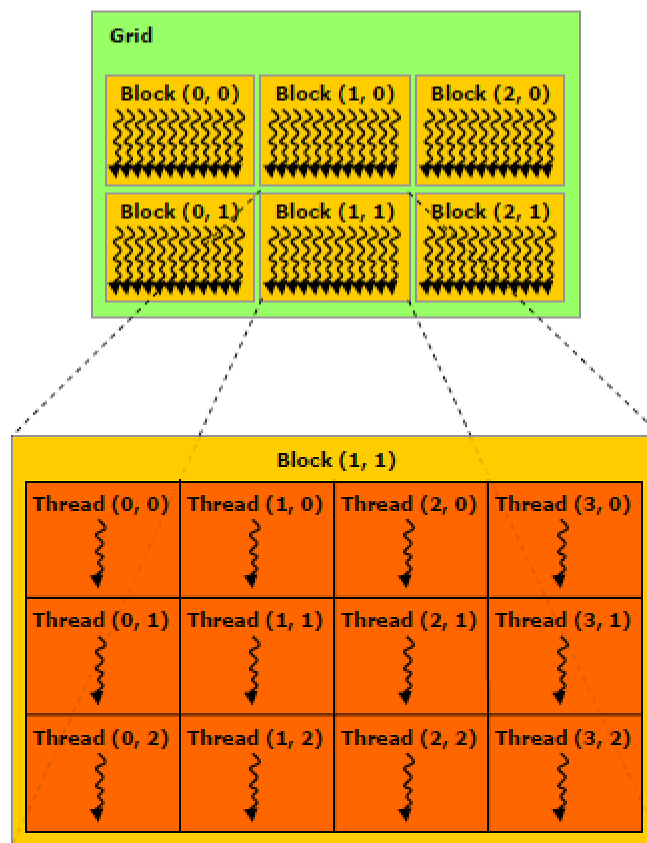
GPU stands for graphics processing unit that is used to produce graphics in computer games, modeling in geosciences/chemistry/biology, image processing, and other graphics-related tasks. GPU is capable of performing trillions of instructions per second which is a huge processing power. A new trend that has started not very long ago, is utilizing the fast processing power of the GPU to parallelize the tasks assigned to the CPU. This trend is known as GPGPU (general purpose graphics processing unit). In our research we investigate the performance of parallelized matrix multiplication on the CPU and GPU. We then propose a hybrid model to do matrix multiplication using both CPU and GPU considering all of the advantages of each processing unit. In this research project we used C programming language to program on the CPU and nVIDIA's CUDA (Compute Unified Device Architecture) to program on the GPU (GeForce 8600 GT).

1. Overview of Research

In this project we used Intel's Quad Core processor and nVIDIA's 8600 GT GPU that has 4 multiprocessors and 255 MB of global memory. First we wrote an algorithm to do a simple matrix multiplication using pthreads on the CPU. We then did matrix multiplication using pthreads and blocking to have a high cache hit. After this we used the same blocking algorithm combined with the architectural features on the GPU to multiply two matrices on the GPU. In all the three experiments we recorded the results. The performance results of the GPU are drastically better than the CPU, which we attribute to the optimized GPU architecture for matrix multiplication. However the GPU has a big limitation: the dedicated memory that it has is very small and this will prevent the multiplication of large matrices (over 4000x4000). We propose a hybrid algorithm that splits a huge matrix into smaller matrices on the CPU and then ports these small matrices onto the GPU one by one and computes the product. It also computes some of the smaller matrices onto the CPU to save time. This produces a GPU scheduler that leverages the architecture and performance of the GPU without the drawbacks.

2. CUDA (Compute Unified Device Architecture)

CUDA is essentially a small set of extensions to the C programming language that enable a straightforward implementation of parallel algorithms. As we know GPU is a massive parallel processor that has many cores on it and GPU Computing requires a programming model that can efficiently express that kind of parallelism, most importantly, data parallelism. CUDA implements such a model. Parallel portions of a program are called Kernels which are nothing but functions. These functions can be executed in parallel due to the multi-core architecture that the GPU provides. A key point to remember here is that only one kernel is executed at a time so it is quite not the same as CPU where you can have different functions running at the same time. CUDA threads are extremely lightweight threads as compared to CPU based pthreads that require more overhead to create them and for context switching. CUDA has thousands of threads running at the same time in contrast to CPU where you have few threads running at the same time, this increase the parallelism to a great extends. Below is the figure that will help in understanding how a kernel is executed in parallel using CUDA.



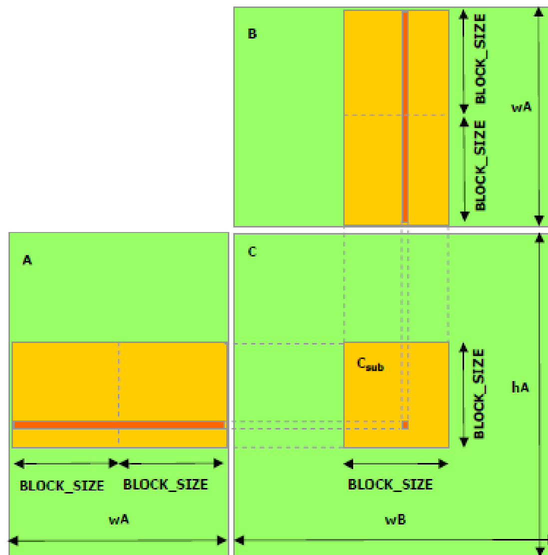
A Kernel or a function is represented by a grid of blocks, where each block has many threads running in itself in parallel. Each thread is executed by a physical thread processor and in a block there are many thread processors. Each block is executed by a

multiprocessor and the hardware does the scheduling for blocks between the available multiprocessors. The nVIDIA architecture also has a very fast shared memory for each block of size 16KB, this memory is exclusive to threads in that block. Every block on the grid is assigned a block ID that has two components, the x and the y components and similarly every thread in a block also has two components the x and y component. CUDA allows the user to define grids of up to dimension of three means we can have at most x, y and z components. For our matrix multiplication we used a grid of dimension two since our matrices were two dimensional matrices. All threads are allowed to access the global memory of the GPU which is 255MB in our project.

3. Parallelized Matrix Multiplication Algorithms

- CPU Based

  Here we used p-threads to achieve parallelized matrix multiplication. An example would be an ideal way to explain this algorithm. Assume we have three matrices A, B and C. A [**hA** x **wA**], B [**wA** x **wB**], C [**hA** x **wB**] = A x B. We divide the resultant matrix into blocks of size 64 x 64 and compute block by in parallel. Since we had four cores on our machine, we spawned four threads to compute four blocks in parallel. The diagram below illustrates the idea.



  Every block on the matrix C is computed as sum of the products of sub-blocks of matrix A and matrix B. The idea of using blocks is to increase the locality of reference or to have a high cache hit. A regular matrix multiplication algorithm takes one row of matrix A and multiplies each of its elements with the corresponding elements in the corresponding column of matrix B. We see that doing so will result in a good number of cache miss on matrix B since we traverse the column in B vertically but the memory addresses are aligned horizontally. So

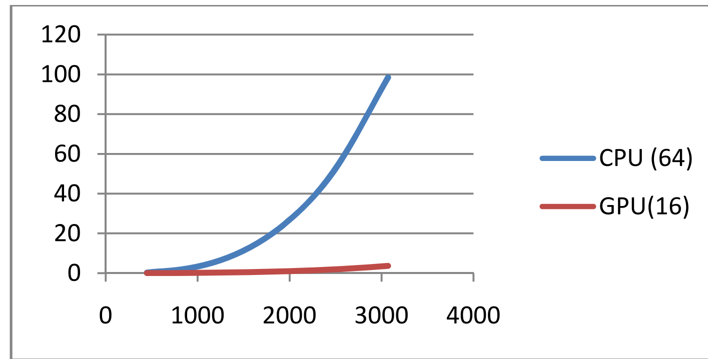
we made a small optimization at this stage. Assume you are multiplying two blocks A and B and the resultant block is C, then every element in matrix C is as follows $C_{ij} = \sum_k A_{ik} \cdot B_{kj}$. What this means is that we take one element from the column 'a' in block A and multiply it with all the elements in row 'a' of block B and we get 'j' different elements that we store in block C as $C[i,j] := C[i,j] + A[i,k] \cdot B[k,j] \ (0 \le j \le n-1)$. This step will be illustrated with slides during the presentation for more clarity.

- GPU Based

Same algorithm was followed on the GPU as well but with a major change in the number of threads being used. We divided the whole operation into CUDA grid where each block refers to an actual block on the matrix C of size 16. Csub = $\sum Asub_i * Bsub_i$ (refer to the previous illustration). At one instance in time one sub matrix of A and B is loaded from global memory into the shared memory of the cache by 32 different threads and then each thread multiplies one row and its corresponding column to get one resultant element of matrix C. Each block on the GPU has 32 physical threads that execute parallel piece of code and each block can have a maximum of 512 threads (that will involve context switching). As we see even after using the same algorithm as we used for the CPU based version, it is much more parallel due the architecture that is provided by nVIDIA's GPU.

4. Results

Matrix Dimension	CPU(64)	GPU(16)
448	0.31s	0.10s
960	2.99s	0.20s
1472	10.72s	0.47s
1984	26.24s	1.03s
2496	52.26s	1.94s
3072	98.5s	3.67s



Matrix Dimension	GPU(22)
484	0.13s
990	0.33s
1496	0.89s
2090	1.50s
2596	4.49s
3080	7.64s

- Explanation

GPU(22) means the block size was 22 and GPU(16) means the block size was 16. Block sized 16 will have 16*16 threads and block size 22 will have 22*22 threads. As predicted GPU performs faster than the CPU for the same matrix multiplication algorithm used on either sides. The reason being highly parallel architecture of the GPU. However as we increase the block size of the GPU to 22 we notice that the performance decreases as compared to GPU with block size 16, this is because in a larger block size there are more threads operating and therefore there is more context switch. On each block 32 physical threads actually run at a time. A second reason is due to the width of the multi-processor which is 32, which is beneficial in the case of block size of 16 as it can load two rows from the block (16 + 16) and run them by occupying its entire width but in the case of block size 22 only one row of size 22 is executed and 10 threads out of the 32 physical threads get wasted. Hence we do not utilize the architecture completely.

5. Limitations

p-threads are expensive to create and they take more overhead in context switching than GPU threads. Despite GPU having a better architecture for parallel computing, especially for matrix multiplication, it has much less dedicated memory space (255 MB in our case)

compared to the RAM in higher-end computers. From the results it is clear GPU is very fast but has less memory in contrast to CPU which is slower but has more memory.

6. Proposal for a better performing scheduler

Given two matrices A and B of equal dimensions NxN so that the resultant matrix C = AxB is also of dimension NxN. We assume square matrices just for the sake of clarity. We need to design the algorithm such that we use GPU for maximum parallel computation and CPU to do the scheduling of these computations keeping in mind the available memory on the GPU. From what we have researched there is no method of directly finding how much memory is available of the GPU except for the fact we can try to allocate memory and see if it is allocated or not. We slice the matrix C into slices such that to compute each slice the memory that the GPU will require is available on it and if not we then cut these slices into blocks using binary search method till we find an optimum block size. Optimum means using the block size such that we utilize the max available memory on the GPU. Sometimes one might think that this could affect other process running on the GPU but in CUDA only one kernel (function/process) runs at a time. In concurrent to this we assign one slice to the CPU which breaks each slice into blocks of size 448x448 and computes them individually so that at any point in time if the GPU has finished computing rest of the matrix then we can assign the existing blocks in the slice that CPU is computing to the GPU. We choose the size 448x448 because for this size the cost on the CPU and GPU is almost the same as from our experiments. A second optimization that we could possibly make is use the OpenMP framework for doing the computations in the CPU as it has slightly better performance than p-threads.

7. References

[1] nVIDIA's CUDA Documentation http://www.nvidia.com/object/cuda_develop.html

[2] nVidia CUDA Software and GPU Parallel Computing Architecture http://www.ece.wisc.edu/~kati/fpga2008/fpga2008%20workshop%20%2006%20NVIDIA%20-%20Kirk.pdf