Bank Conflict Latency of Convolutions on GPUs

Yuan Liu and Ye Htet

December 2021

Abstract

In this paper, we study the bank conflicts of convolution algorithms on the GPU and demonstrate an asymptotic analysis of the corresponding latency on shared memory. In particular, for a convolution with a naive approach using a $f \times f$ kernel on an image of size n, the memory access latency is shown to be $O(f^2\frac{n}{w} + l\frac{n}{w})$ with width w and latency l, using the DMM model. Our approach shows us how increasingly difficult it would be to prove similar results when the convolution no longer follows a naive approach.

1 Introduction

There has been a great number of research done on the GPU in the past decade as the GPU has extensively improved from just a graphics processor into a fully general purpose, programmable, and manycore processor[2][4][10][11][12].

However, despite these works, there has only been experimental results and very few, if not any, attempts at analyzing algorithms on GPUs[1][3][6].

Hence, this paper attempts to add to the work of theoretical analysis of algorithms on the GPU. In particular, we will be looking at how filters used in convolution can degrade the parallelism on the GPU. Our analysis shows how exactly performance becomes restricted.

2 Background

2.1 GPU Application

The GPU has grown greatly in use over the years. Its dedicated processors has been replaced by programmable processors and with the ability to perform even more precise computations, starting out from indexed arithmetic to double precision floating-point, the GPU can now be used for a great number of applications.

There is General Purpose computation on GPU(GPGPU), which is the use of GPU to compute non-graphics jobs while still using the traditional graphics pipeline.[11] But, by abandoning this model, programming parallel and general computations can been done entirely without using the graphics pipeline and is known as GPU Computing. Finally, there is Compute Unified Device Architecture(CUDA) which is a platform that allows programming scalable and general parallel computations for the GPU, turning the once specialized use of the GPU to be easily accessible for the more common programmers.

2.2 GPU Architecture

We shall demonstrate a simple model of the GPU necessary for our work. The GPU consists of streaming processors organized into multithreaded streaming multiprocessors (SM) each of which has their own shared memory that the processors can access directly. In addition, there is a slower and larger global memory that all the SMs has access to. The SMs use the single-instruction multiple-thread (SIMT) architecture to manage and run groups of w numbers of threads called warps in parallel.

The global memory allows communication between different SMs and to optimize utilization of this memory, access must be coalesced. This means that the threads of a warp must access consecutive blocks in global memory allowing all the threads to get their own block in parallel.

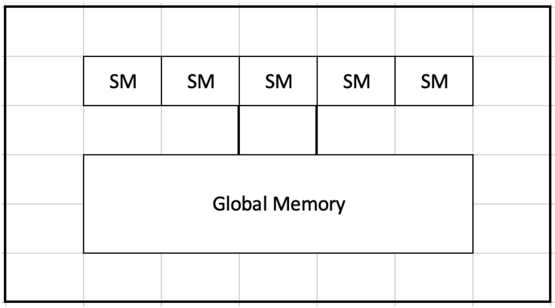


Figure 1: Streaming Multiprocessors connected to Global Memory in a GPU

Shared memory on the other hand is smaller and faster than the global memory and is only accessed by the different streaming processors in a single SM. This memory is further divided into memory banks. When two or more threads access the same bank, the access is serialized, losing parallelism, with the exception of the threads ac-

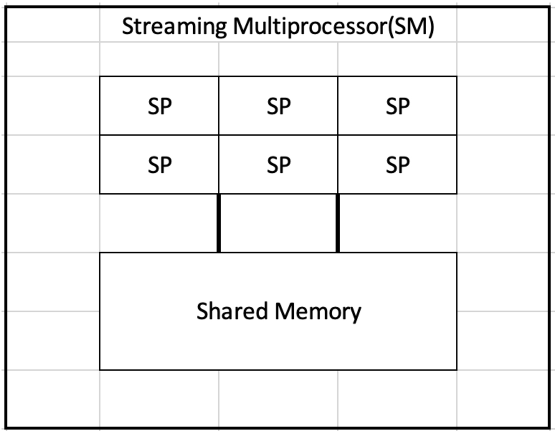


Figure 2: A Single Streaming Multiprocessor

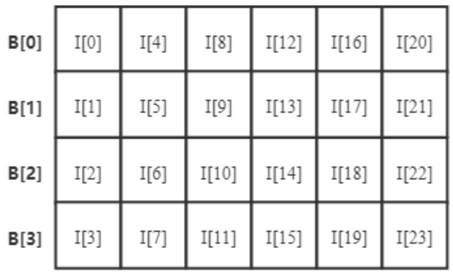
cessing the exact same cell in the bank, in which, the cell can be multicasted to all the threads. This issue is called a bank conflict and is one of the cause of performance degradation.

When doing any theoretical analysis of parallel computing on an architecture, there are several models that can be used to provide the necessary abstraction. The survey[9] by Navano et al. explains the different type of models namely. the Parallel Random Access Machine (PRAM), Parallel Memory Hierarchy (PMH) and Bulk Synchronous Parallel(BSP). However, it turns out in our case that the Discrete Memory Machine model introduced by Nakano[8] matches the needed features for our analysis.

2.3 Discrete Memory Machine Model(DMM)

Discrete Memory Machine Model(DMM) is a practical GPU architecture model for memory complexity analysis on GPUs, which captures the essential feature of memory access of NVIDIA GPUs [8]. Let's now introduce DMM Model of height w, width s and latency l. In DMM Model, a number of cores share a shared memory. The shared memory consists of w banks, each of which is a row and has s memory cells. The model has 3 major features about shared memory. First, data is stored in column-major order in shared memory. Let's say I is an array and $B[j]$ is the j^{th} bank. Then all $I[iw+j]$, where $i = 0, 2, 3, ..., s-1$, is in bank $B[j]$. Apparently, $I[k]$ will reside in $B[k \; mod \; w]$.

Secondly, there are bank conflicts in shared memory. memory cells in different banks can be accessed simultaneously in one time unit. However, access to memory cells in the same bank can cause bank conflicts. Because of that, memory cells in the same bank cannot be be accessed in parallel in a time unit. Such memory cells have to be accessed by contiguous memory requests. For example, in figure 3, access to $I[0], I[2], I[5], I[11]$ can be done in parallel in just one time unit. But access to $I[0], I[4], I[5], I[11]$ has to be finished by two contiguous memory requests, of which the first one access $I[0], I[5], I[11]$ and the second one access $I[4]$.



B[0]	I[0]	I[4]	I[8]	I[12]	I[16]	I[20]
B[1]	I[1]	I[5]	I[9]	I[13]	I[17]	I[21]
B[2]	I[2]	I[6]	I[10]	I[14]	I[18]	I[22]
B[3]	I[3]	I[7]	I[11]	I[15]	I[19]	I[23]

Figure 3: Bank Conflict Example

Lastly, in DMM Model, contiguous memory requests are processed in pipeline manner. And it takes l time units to complete each memory request. Since all memory requests will be processed in a pipeline fashion, $f + l - 1$ time units are necessary to complete f contiguous memory requests. As shown in figure 4, the first memory request starts at time 0 and ends at time $l - 1$. The second one starts at time 1 and ends at l. So for f memory requests, the first one starts at time 0 and the last one ends at time $f + l - 2$. The total time to complete them is $f + l - 1$.

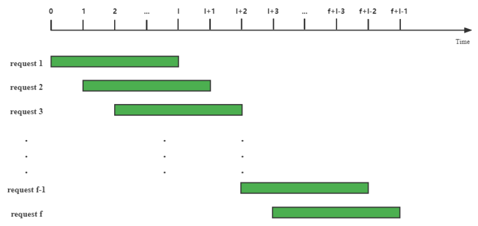


Figure 4: pipeline of memory requests

There are two memory access patterns in shared memory: contiguous access and stride access. For contiguous access, the latency to access an array of size $n (n = w \cdot s)$ is $O(\frac{n}{w} + \frac{nl}{p})$. For stride access, the latency is $O(\frac{nG}{w} + \frac{nl}{p})$, where p is the number of processors in an SM unit and G is the greatest common divisor of s and w. More details and analysis are included in [8].

2.4 Convolution and Filters

The process of convolution stemmed from the area of computer imaging. A simple convolution uses a filter or kernel such as the mean filter which is an $f \times f$ cells square box all containing the value 1. We then place this filter on top of the pixels on an input image and compute the sum of all the multiples of all the overlapping cells and pixels. The result of this sum is outputted back to the pixel at the center of the kernel on a different memory location to avoid contaminating the original image as the kernel now has to continue computing the same output for every pixel in the image. For our mean filter or kernel, the resulting image will be a burred image. An example convolution using a general filter is demonstrated in this figure 5.

Similar to the mean filter we can place in arbitrary values in the cells of the kernel to give interesting outputs after a convolution or the computation. Some simple and notable filters are Median, Gaussian, and Sobel filters, each used for different purposes. A more involved use of filter is in edge detection such as the canny edge detection that makes use of the Sobel filter and this algorithm has been shown interest to be implemented on the GPU as described in this paper[7].

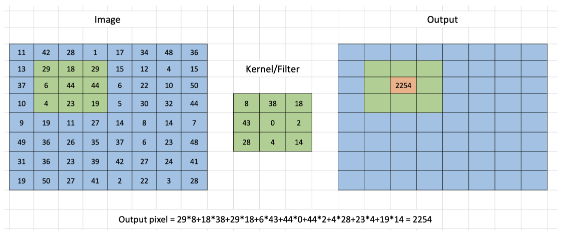


Image
Kernel/Filter
Output
2254
Output pixel = 29*8+18*38+29*18+6*43+44*0+44*2+4*28+23*4+19*14 = 2254

Figure 5: A Convolution

3 Related Works

We drew inspiration from the work of Dr. Nodari Sitchinava from the University of Hawaii at Manoa. In particular, we studied a paper on bank conflict analysis on batch predecessor search in the shared memory of GPUs.[5] In this paper, Karsin et al. looked at the batch predecessor search (BPS) algorithm.

The paper demonstrated that due to bank conflicts, there was a significant degradation in performance when using a naive parallel binary search algorithm on the shared memory of the GPU. In particular, the performance was $\Omega(w \cdot \frac{Q}{p} \log \frac{K}{w^2})$. In comparison, they provided two new approach: (1) the conflict-free PBS search which is not work optimal, needing $O(w - logw)$ extra operations per query but has $\Theta(\frac{Q}{p} \log \frac{K}{w} + \frac{Qw}{p})$ performance, and (2) the conflict-limited PBS which is work optimal, needing $\Theta(\log K)$ operations per query and has the performance of $O(\frac{Q}{p} \log \frac{K}{w} + \frac{Qw}{p})$.

Similar to this paper, we looked at other bank conflict analysis work done by Dr. Nodari Sitchinava such as the sorting and permuting without bank conflict paper[1]. Again in this paper, Afshani et al. showed a $O(mlog(mw))$ sorting algorithm, a partition algorithm that runs $O(m)$ when $w \leq m$ but in general in $O(mlog_m^3 w)$ and finally a randomized permutation algorithm with expected run time $O(mlogloglog_m w)$ where w is the number of banks in the shared memory based on the DMM model.

Thus, similar to these works, we shall apply bank conflict analysis to convolution on the GPU. Convolution and filters are a widely used technique in computer vision and imaging which has been developed extensively with the use of the GPU. Furthermore, since there has been few theoretical analysis of the convolution, we chose to do our study on this specific algorithm.

4 Our Contributions

In the similar vein with Dr. Nodari Sitchinava's work, we apply similar bank conflict analysis to convolutions. We will demonstrate that for a convolution with a naive approach using a $f \times f$ kernel on an image of size n, the run-time will turn out to be $O(f^2 \frac{n}{w} + l \frac{n}{w})$ with width w and latency l, using the DMM model.

With this result, we shall show that the parallelism can be regained by changing the data transfer pattern between global memory and shared memory, bring down the latency from $O(f^2 \frac{n}{w} + l \frac{n}{w})$ to $O(f \frac{n}{w} + l \frac{n}{w})$. We shall also see that this improvement is with limitations.

5 Latency Analysis of Convolutions

Now we will analyze the memory latency of convolutions on GPUs using DMM Model. Let us first clarify our problem. Given an image block I of size n, a convolution kernel F of size $f \times f$ and a shared memory of w banks and s cells in each bank, we want to analyze the memory latency to do convolution operations on it.

5.1 Two Assumptions

Before we start our analysis, we make two reasonable assumptions.

First, the shape of image block I is aligned with the shape of shared memory. That is, $n = w \times s$. It is reasonable because every time we fetch a block of the image from global memory, we can cut the whole image into blocks whose shapes are the same as that of the shared memory. If not, we can satisfy the alignment by padding the image.

The second assumption is to have the image block I stored in shared memory in a row-major order. For our specific problem, however, it does not matter if the block is stored in a column-major order because for the convolution, we can also transpose our kernel to adapt to the orientation of the image block and it has no influence on both the essence of the convolution and our analysis. Thus for convenience, we just say that the image block is stored in a row-major order.

5.2 Pseudo Code

Algorithm 1: Conv(image $I_{w\times s}$, kernel $F_{f\times f}$)

```
1 for stage $k \leftarrow 0$ to $\frac{n}{w}$ do
2     for round $r \leftarrow 0$ to $f$ do
3         for $u \leftarrow 0$ to $\frac{w}{f}$ do
4             do in parallel: proc(u) access
                $f \times f$ block of $I[r + u \cdot f, k]$;
5         end
6     end
7 end
```

5.3 Access to one $f \times f$ block

Based on our survey, most convolution algorithms on GPU assigns each pixel to a thread, which means each thread accesses a $f \times f$ neighborhood of one pixel. Since there are n pixels in shared memory, n neighborhoods want to be accessed simultaneously. However, some of these n neighborhoods are overlapping with each other. Therefore, these overlapping ones have to be done in a pipeline manner or sequentially. Later on, we will talk about which threads can be done in parallel and which cannot be done. But before that, we first talk about how much latency it takes to complete memory access to just one $f \times f$ neighborhood for one thread.

For one thread, as shown in figure 6, memory access to an $f \times f$ block has to be decomposed to f contiguous memory access requests, each of which accesses one of the f columns in the block. Because each column has memory cells in the same banks, there are bank conflicts and thus these f requests will be processed in a pipeline manner. According to section 3.3, we know that these f contiguous requests will be finished in $f + l - 1$ time units.

5.4 Access to Blocks in Parallel

However, as we said, not all threads can access their blocks in parallel. Once one block is accessed, any other blocks who are overlapping with it have to be stashed in the pipeline and processed later. Also, any other blocks whose cells are in the same banks with the current block also cannot be processed in parallel with it because of bank conflicts. As shown in figure 7, when block 1 is be accessed, neither block 2 nor block 3 can

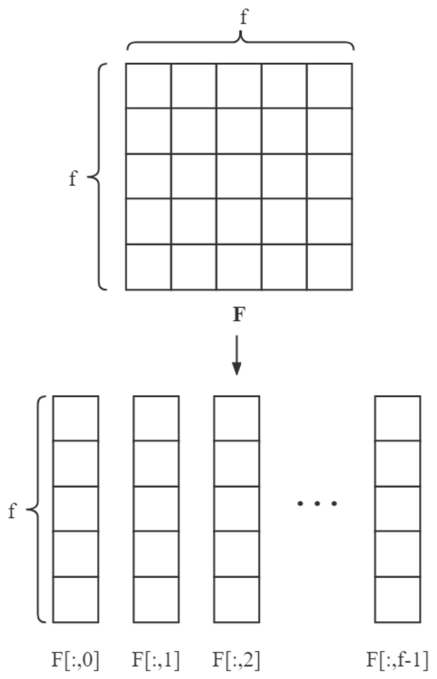


Figure 6: Decomposition of a convolution kernel

be done in parallel with block 1. Both of them need to access the same banks as block 1 does. But block 4 can be done with block 1 simultaneously. Therefore, we can conclude that vertically contiguous convolution blocks can be accessed in parallel.

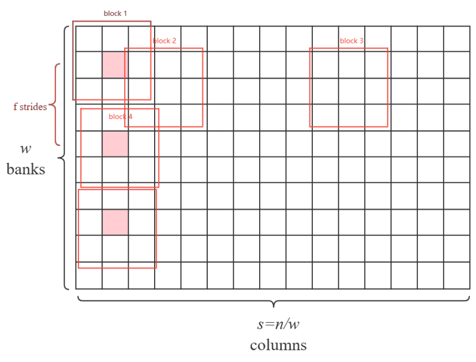


Figure 7: Memory access pattern in convolution algorithms

Now the whole picture is as figure 7 shows. There are $O(\frac{w}{f})$ blocks accessed in parallel, which means $O(\frac{w}{f})$ pixels in one column are processed. We call such a step a "round". In each round, all threads will first access the first column of their own block in parallel. And then they access the second one, the third one until the $(f-1)^{th}$ column in pipeline manner. As shown previously, the latency to finish one round is $O(f + l - 1)$.

When one round is completed, all the threads will move their block one cell down vertically and then access the neighborhood of their new pixels. To finish processing all the pixels in one column of shared memory, $O(f)$ rounds are needed. And we call the time to finish pixels in one column a "stage". Since each round includes f contiguous memory requests, one stage will have f^2 contiguous requests. So the latency for one stage is $O(f^2 + l - 1)$. And s stages are necessary to process all pixels in the shared memory. However, because of barrier synchronization in GPU, memory requests between different stages have to be sequential rather than in pipeline manner. Therefore, the total time is $O(s \cdot (f^2 + l - 1)) = O(f^2 \frac{n}{w} + l \frac{n}{w})$.

5.5 Result Analysis

As we can see, there are two terms in our result, of which one is related to f and the other one is related to l. Actually, these two terms clearly show the influence on memory latency because of bank conflicts and memory request time. If $f^2 > l$, then the memory latency is $O(f^2 \frac{n}{w})$. And we can explain this result from a more intuitive perspective.

First, when $f^2 > l$, it means the pipeline of memory requests, which is of length l, can be filled up with f^2 memory requests. It is actually the same as "CPI" concept in computer architecture area. Processing instructions in pipeline manner can reduce the average time of each instruction. Similarly, putting memory requests in pipeline manner can hide the latency of each request. Thus the result is $O(f^2 \frac{n}{w})$.

Secondly, $O(f^2 \frac{n}{w})$ is a reasonable result because we have n pixels in total. Each pixel will have to be accessed f^2 times because there are f^2 different blocks containing the pixel. Furthermore, there are bank conflicts among these f^2 blocks since they are overlapping each other. So we have to access $f^2 \cdot n$ memory cells in total. And because of bank conflicts, we can access at most w memory cells in parallel in each timed unit.

6 Increase Parallelism

It is clear that bank conflict is the barrier that does harm to the parallelism. And the overlapping between different convolution blocks is the major source of bank conflicts. As shown in figure 8, pixel $I[1]$ and $I[2]$ cannot be processed simultaneously because of their blocks overlap with each other. If there is no overlapping, we then can process the two pixels in parallel.

Therefore, instead of simply fetch a block of image from global memory, we transfer the data into shared memory by rows. For some row j

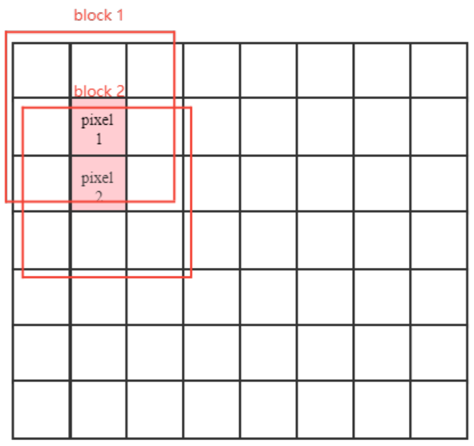


Figure 8: bank conflict between neighborhoods of two pixels

of the image, we transfer the $\lfloor \frac{f}{2} \rfloor$ prior rows and $\lfloor \frac{f}{2} \rfloor$ subsequent rows of row j into shared memory. Then we move to row j+1 and do the same thing until we fill up the shared memory or finish all rows of the image. As figure 9 shows, given $f = 5$, for row 3 we put row 1,2,3,4,5 into shared memory. And then for row 4, we put row 2,3,4,5,6 into shared memory. Doing so, all pixels in the same column can be processed in parallel without bank conflict. In other words, we can finish a stage in just one round rather than f rounds. So our new result is $O(f \frac{n}{w} + l \frac{n}{w})$, where one f is eliminated. The remaining f in the result is because of horizontally contiguous memory requests, which cannot be eliminated in our method.

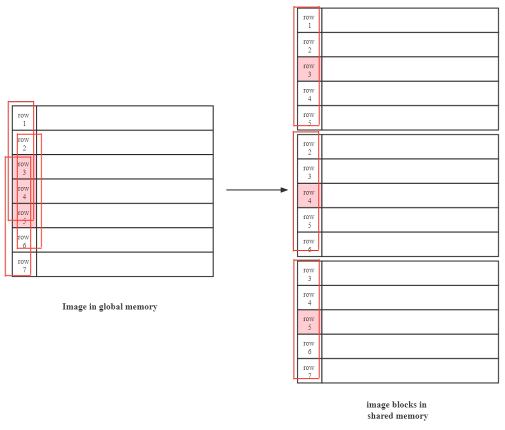


Figure 9: Eliminating bank conflicts between rows

There are some changes we should be aware of when comparing the original method and this new method. In the original method, each memory cell will be accessed by f^2 different memory requests. And n pixels will be processed in shared memory(or in each SM unit) after the whole process. In our method, each grid are only accessed f times. And $\frac{n}{f}$ pixels in total will be processed after the whole process. It may be confusing that in our method number of processed pixels in each Sm is even smaller than before. However, we can use more SM units to run more blocks in parallel. And that's exactly how we increase the parallelism.

However, we acknowledge that our method has some limitations. First, the improvement is based on an assumption that we have enough SM units in a GPU. Precisely, the number of SMs now is f times as the number of SMs before. If Sm units are not sufficient, the parallelism will decrease. Secondly, though our method can reduce the latency of shared memory, it may increase the latency on global memory because a more complicated data transfer pattern between global memory and share memory are used in our method. Also, since we are replicating more than one copy of each row, the amount of data is greater than before. So it takes more time to transfer data between global memory and shared memory.

7 Conclusion

In this project, we used DMM model and analyzed the bank conflict latency of convolution on GPUs. We have shown that the convolution with $f \times f$ kernel on an image of size n can be done in $O(f^2\frac{n}{w} + l\frac{n}{w})$ time on the DMM model with width w and latency l. And the latency term can be hidden if $f^2 > l$.

Also, we tried to increase the parallelism by changing the data transfer pattern between global memory and shared memory. In our method, the latency reduces from $O(f^2\frac{n}{w} + l\frac{n}{w})$ to $O(f\frac{n}{w} + l\frac{n}{w})$. However, our method has limitations, which are requirement for more hardware resources and increase of global memory latency.

After this project, we found how hard it is to do theoretical memory analysis on GPUs. Our analysis is based on some assumptions that help us simplify the problem. It would be even more difficult to analyze what's happening on GPU when the memory access pattern gets more complex. That may be the reason why more papers prefer to convince readers by experimental results. However, it is necessary to do some theoretical analysis before we implement an algorithm on GPUs because implementation takes more time and is more expensive.

With machine learning on GPU more and more popular, we think theoretical analysis of machine learning algorithms on GPU is really an interesting area to research on.

References

[1] Peyman Afshani and Nodari Sitchinava. Sorting and permuting without bank conflicts on gpus. In *Algorithms-ESA 2015*, pages 13–24. Springer, 2015.

[2] Dmitri I Arkhipov, Di Wu, Keqin Li, and Amelia C Regan. Sorting with gpus: A survey. *arXiv preprint arXiv:1709.02520*, 2017.

[3] Kyle Berney and Nodari Sitchinava. Engineering worst-case inputs for pairwise merge sort on gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1133–1142. IEEE, 2020.

[4] Gaurav Hajela and Manish Pandey. Parallel implementations for solving shortest path problem using bellman-ford. *International Journal of Computer Applications*, 95(15), 2014.

[5] Ben Karsin, Henri Casanova, and Nodari Sitchinava. Efficient batched predecessor search in shared memory on gpus. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pages 335–344. IEEE, 2015.

[6] Ben Karsin, Volker Weichert, Henri Casanova, John Iacono, and Nodari Sitchinava. Analysis-driven engineering of comparison-based sorting algorithms on gpus. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 86–95, 2018.

[7] Yuancheng Luo and Ramani Duraiswami. Canny edge detection on nvidia cuda. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8. IEEE, 2008.

[8] Koji Nakano. Simple memory machine models for gpus. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 794–803. IEEE, 2012.

[9] Cristobal A Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.

[10] Umesh Nayak, Rajeev Pandey, and Mr Uday Chourasia. A survey paper on analyzing shortest path algorithms on gpu using opencl.

[11] John Nickolls and David Kirk. Graphics and computing gpus. *Computer Organization and Design: The Hardware/Software Interface, DA Patterson and JL Hennessy, 4th ed., Morgan Kaufmann*, pages A2–A77, 2009.

[12] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. Scan primitives for gpu computing. 2007.