Efficient CUDA

CODY COLEMAN and DANIEL KANG, Stanford University, USA

Recent work in deep learning has created a voracious demand for more compute cycles, but with the slowdown of Moore's law, CPUs cannot keep up with the demand. Thus, attention has turned to massively-parallel hardware accelerators, ranging from new processing units designed specifically for deep learning to the repurposing of existing technologies like Field-programmable gate arrays (FPGAs) and graphics processing units (GPUs). While these accelerators can provide immense speedups over traditional CPUs in certain domains, application programmers must take great care in optimizing code. In this work, we focus on GPUs and explore how 1) memory access patterns, and 2) memory abstractions can affect the performance of massively-parallel hardware accelerators, in the context of matrix multiply. We show that by optimizing memory access patterns, we can achieve over 15× speedups and choosing the wrong memory abstractions can lead to a 40× slowdown. Thus, we show the importance of memory in programming modern GPUs.

CCS Concepts: • **Programming Languages**;

ACM Reference Format:
Cody Coleman and Daniel Kang. 2017. Efficient CUDA. 1, 1 (December 2017), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SUMMARY

In this work, we analyze the performance considerations of matrix multiplication on GPUs, as a case-study of workloads on massively-parallel hardware accelerators. GPUs and, more generally, SIMD architectures have different performance characteristics (e.g. lower clock cycle but many more cores) than standard CPUs and so the performance considerations dramatically change. We analyze 6 versions of matrix multiplies, which represent a variety of optimizations (primarily dealing with memory efficiency) that highlights some performance considerations in GPU programming. We also study the three memory abstractions CUDA offers. We find that seemingly simple decisions can make up 15× and 40× difference in performance for memory access and abstractions respectively.

Equal work was performed by both project members.

2 INTRODUCTION

In recent years, deep learning has delivered incredible results in applications ranging from classifying images [He et al. 2016] to ad placement [Bours 2017]. These results, coupled with the plateau of clock speeds [Eeckhout 2017] have led to an incredible demand for more floating point computation. A variety of new hardware accelerators have been developed and repurposed to address this growing computational, including Google's Tensor Processing Unit [Jouppi et al. 2017], Microsoft's Project Brainwave built on FPGAs [Blog 2017], Graphcore's Intelligence Processing Unit [Graphcore 2017], and NVIDIA's Graphics Processing Unit (GPU) [NVIDIA 2016, 2017]. Given the trends in deep learning, we expect these demands to only increase.

However, these hardware accelerators are designed with very different applications in mind. For example, consider the GPU. GPUs were initially designed for graphics processing, but are now primarily used for single instruction, multiple data (SIMD) applications, in which many processing units apply the *same* operation on many pieces of data simultaneously. The NVIDIA P100 has 3584 CUDA cores and the NVIDIA V100 has 5120 CUDA cores, which allows these accelerators to achieve up to 9.3 and 15 tera-floating point operations per second (FLOPS) [NVIDIA 2016, 2017]. However, each individual CUDA core has limited performance: the V100 has a clock speed of 1530 MHz, compared 2666 MHz for some Intel CPUs [NVIDIA 2017]. In order to realize the potential of GPUs, programs must divide work into many small chunks that can execute in parallel.

Moreover, the CPU still serves as an intermediary between the application and the GPU. Data must be transferred from the CPU to GPU before computation can begin, and results must be transferred back to CPU when the computation finishes. As a result, in addition to normal memory management, extra care must be taken to efficiently transfer data between devices. In case of NVIDIA GPUs, there are several abstractions to simplify memory management between devices.

In this work, we explore performance considerations of massively parallel architectures, specifically in the context of GPUs. We analyzed various computational and memory optimizations of matrix multiply, which is a critical operation in deep learning, and more generally, in scientific computing.

The remainder of the paper continues as follows. The Background section overviews the memory hierarchy and abstractions of modern NVIDIA GPUs, followed with a brief review of matrix multiplication. The Methods outlines our system set up and the 6 variants of matrix multiplication considered. The Results section looks at each of these variants for 3 different memory management systems and a variety of matrix sizes. We then conclude with a summary of our work and key takeaways for efficient CUDA programming.

3 BACKGROUND

3.1 GPUs

While GPUs were initially designed for graphics processing, they have been repurposed for scientific computing [top 2017; Fan et al. 2004] and, more recently, deep learning [Jia et al. 2014]. Modern GPUs have many more cores that can support thousands of concurrent threads. However, these cores are simpler and are primarily used for floating point operations. Additionally, the total bandwidth for on-GPU memory is typically much higher the bandwidth from CPU to RAM [Fan et al. 2004]. Given these differences in design, GPUs and CPUs have a variety of trade-offs, with GPUs excelling in computations that have a significant amount of similar operations

Authors' address: Cody Coleman; Daniel Kang, Stanford University, 353 Serra Mall, Stanford, CA, 94305, USA, cody@cs.stanford.edu, ddkang@stanford.edu.

© 2017 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in , https://doi.org/10.1145/nnnnnnn.nnnnnnn.

Fig. 1. The P100 SM architecture. Each SM consists of 56 cores and has its own L1 cache and shared memory. This diagram is taken from [NVIDIA 2016].

and high data-locality. This style of parallelism is known as Single Instruction, Multiple Data (SIMD).

As GPUs have limited use, they are used as *accelerators*. While there are many performance considerations in programming with hardware accelerators, we focus on memory aspects in this work.

3.1.1 GPU memory abstractions. The programming abstraction of choice for NVIDIA GPUs is CUDA, which offers three methods for memory management: manual memory management, unified virtual addressing (UVA), and unified memory [NVIDIA 2016].

As the CPU and GPU are separate physical devices, each with it's own memory, data must be transferred to and from the GPU (typically through PCI-E). In manual memory management, this was the abstraction offered to CUDA programmers, i.e. no abstraction. Thus, data must be manually copied to and from the GPU. This abstraction was the only one offered until CUDA v4.

UVA is an abstraction that allows the CUDA programmer to program as if the GPU and CPU shared a single address space [Schroeder 2011]. UVA works as follows: if the accessed memory does not reside on the current device (i.e. a given GPU), it will copy the data to the current device. Concretely, if the GPU attempts to access data on the CPU, under UVA, the data will be transferred to main memory before being accessed. Additionally, UVA allows "zero-copy" memory, which can access host memory via the PCI-E link. While this provides a simple way of addressing memory, it does not provide any performance improvements over manual memory management. For example, the zero-copy abstraction is limited by PCI-E performance.

CUDA has offered unified memory from v6 (and improved it in v8) [NVIDIA 2016]. Unified memory allows applications to use a single pointer in both CPU and GPU functions. From CUDA v8, unified memory allows for virtual addressing, which will automatically page-fault on non-resident memory. Virtual addressing allows for seamless transfer between GPUs or from the CPU and GPU. This abstraction is continuously being improved by NVIDIA [NVIDIA 2017].

3.1.2 GPU memory hierarchy. The GPU memory hierarchy differs from the CPU memory hierarchy. First, we discuss the layout of a GPU. NVIDIA GPUs are broken down into *streaming microprocessors* (SMs), which contain multiple cores. Each core runs a *warp* of multiple threads. In the P100 architecture, there are 64 cores per SM and 56 SMs per GPU [NVIDIA 2016].

Each SM has its own L1 cache and 64 KB of *shared memory*. In GPUs, the L1 cache is also known as a texture and is read only. The L2 cache is shared between SMs and lies between the SMs and global memory [NVIDIA 2016].

The SM architecture and hierarchy is diagramed in Figure 1.

3.2 Matrix multiply

While matrix multiplies are a simple operation, they are extremely important in scientific computing and even deep learning. For example, multi-layer perceptrons (MLPs) currently make up a plurality of the deep learning workload in Google's infrastructure [Jouppi et al. 2017], and the bulk of the computation in MLPs are in matrix multiplies.

The naive matrix multiply algorithm requires $O(N^3)$ multiply-adds and $O(N^2)$ memory accesses. However, naively performing the matrix multiply results in highly inefficient data access patterns, which significantly degrades performance (which we show in the results). In this work, we explore a variety of optimizations, mainly from efficient memory access, of the matrix multiply.

While there are faster matrix multiply algorithms (notably Strassen's algorithm), we do not study these in this work [Strassen 1969]. Tuning Strassen's algorithm is a complicated process, with "conventional wisdom" up until 2016 noting that the naive matrix multiply is competitive with Strassen's algorithm [Huang et al. 2016].

3.3 Prior work and relationship to PL

Matrix multiplication is a highly studied problem and NVIDIA provides a highly optimized version through CuBLAS. However, its implementation is proprietary. Our goal was primarily to understand the performance characteristics of a popular GPU workload. While we could have decided on another workload, we decided on matrix multiply due to it's popularity.

The CUDA programming framework and it's various memory abstractions offer a balance between usability and performance. Our work explores this trade-off.

4 METHODS

4.1 System setup

All experiments were performed on a server with one NVIDIA P100 GPU and an Intel Xeon E5-2690 (clock speed of 2.60GHz). We used CUDA Version 8.0 and CuBLAS version 2.0.

To time each function, we "warmed up" the function by running it 3 times (this was done to ensure the instructions were transferred to the GPU and the GPU was not in an idle state). Then, the function was run 10 times and the average time was recorded. Correctness was evaluated against CuBLAS SGEMM as ground truth, where we allowed a maximum error of $1x10^{-5}$.

Each function was run with the three different memory management options (manual, UVA, unified).

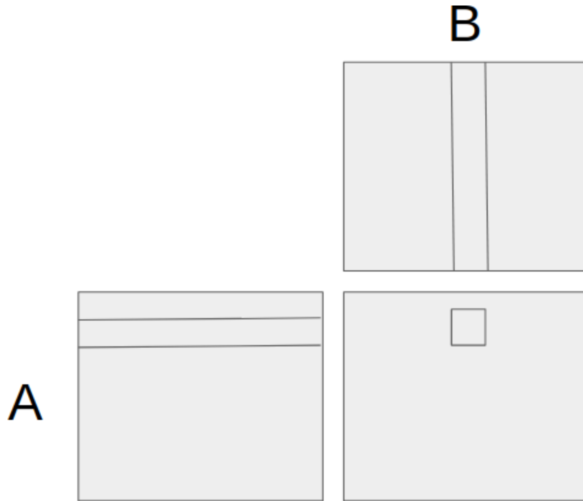

Fig. 2. The naive matrix multiply computation. Each row and column is accessed sequentially.

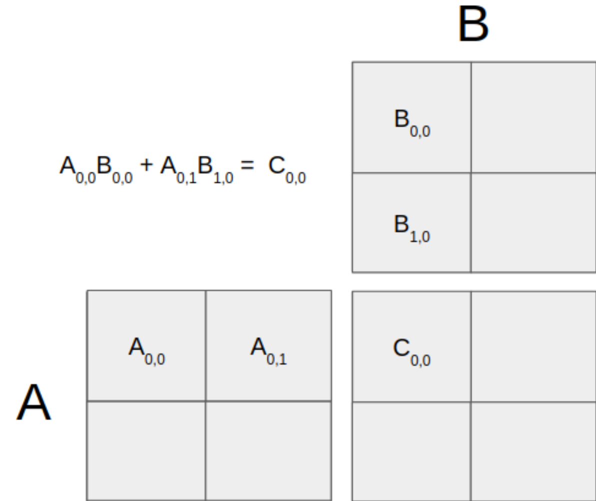

Fig. 3. The tiled matrix multiply computation. The input matrices are broken down into sub-matrices and then multiplied.

The matrix multiplication code was largely inspired by [Wijtvliet [n. d.]]. We wrote the entirety of the testing and timing framework.

4.2 Matrix multiply

We implemented 6 version of matrix multiply, and compared against an optimized CuBLAS matrix multiply provided by NVIDIA.

Throughout, we denote the input matrices as A and B, and the resulting matrix $C = A \times B$. Elements of the matrices are denoted by subscripts, e.g. A_{ij}. To simplify the analysis, we only multiply square matrices. We denote the size as N.

Unless specified, for a matrix of size N we use a block size of 16x16 and threads equal to $N/16$. The thread and block IDs will be denoted as t_x, t_y and b_x, b_y respectively.

4.2.1 Naive matrix multiply. In the naive matrix multiply, each thread computes a *single* element in C. Namely, block b and thread t will compute $C_{b_y \times 16 + t_y, b_x \times 16 + t_x}$. C_{ij} is computed by looping over row A_i and column B_j. A schematic of the computation is shown in Figure 2.

Every element C_{ij} requires exactly N multiply-adds and $2N$ memory accesses. Thus, every multiply-add requires 2 memory accesses. As we see in the results, this dramatically affects performance.

4.2.2 Tiled matrix multiply. In the tiled version of matrix multiply, each thread will still compute a single element in C. However, in the tiled version of matrix multiply, each thread block contains a local shared array and A and B are loaded into the shared memory before being multiplied. We can achieve speedups this way as shared memory can be up to 100× faster than uncached, global memory. A schematic of the computation is shown in Figure 3

Each thread loads one element from A and one element to B for each 16 multiply-adds. We note that the size of the shared array could have been optimized, but we did not explore this optimization.

4.2.3 No bank conflict matrix multiply. Shared memory on NVIDIA GPUs is split into "banks" of memory (a similar construct on the CPU is the idea of a cache line), which can be accessed simultaneously. Thus, loading from b distinct banks will result in b times faster access compared to loading a single bank of memory. However, if multiple threads request access to the same bank of memory, this will cause a *memory bank conflict*, which causes the memory accesses to be serialized.

In the naive tiled matrix multiply, loading elements from global memory will cause a bank conflict, as B is loaded in a column-oriented fashion. This can be optimized by instead loading B in a row-oriented fashion and changing the computation to account for this.

4.2.4 Compute optimized matrix multiply. The above matrix multiply is still a variant of the tiled matrix multiply, in that only the memory loads are modified. However, each multiply-add requires access to shared memory. Instead, we can change the compute pattern by computing *outer products* of sub-matrices. This change decouples the access to B from the access to C, which allows us to store a single element of B and a portion of C in registers.

Concretely, this is done in three steps. First, a submatrix of A is loaded into memory. Then, each thread loads a single element of B and a single column of C in registers. Finally, the outer product is computed and stored.

4.2.5 Unrolled-loop matrix multiply. To further pipeline and reduce branch misprediction and stalling, we can unroll the inner loops for slightly improved performance.

5 RESULTS

5.1 Effects of optimizations

We hypothesized that the optimizations we added would dramatically increase performance over the naive matrix multiply, but be far from the highly optimized CuBLAS version.

The following analysis is done with the results from manual memory management.

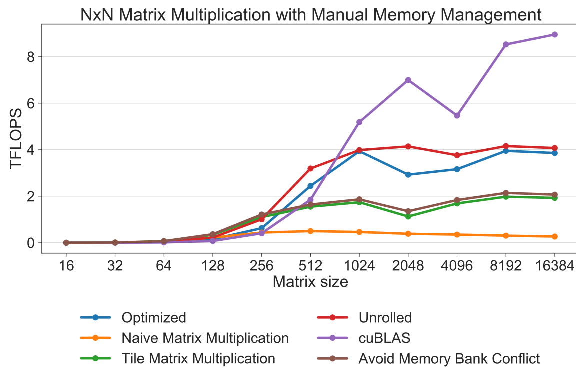

Fig. 4. 6 implementations of matrix multiplication on a NVIDIA P100 GPU for various matrix size using manual memory management. All points are the average of 10 runs and the matrices are read from device memory

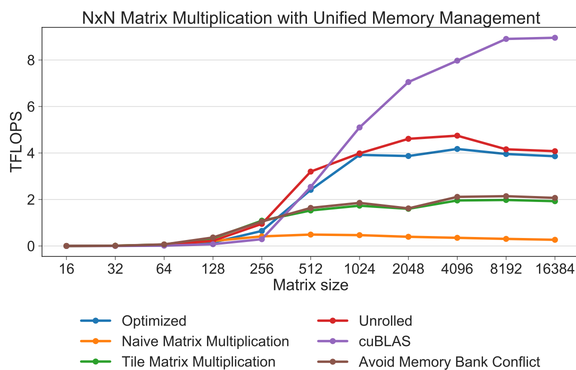

Fig. 5. 6 implementations of matrix multiplication on a NVIDIA P100 GPU for various matrix size using unified memory management. All points are the average of 10 runs and the matrices are read from device memory

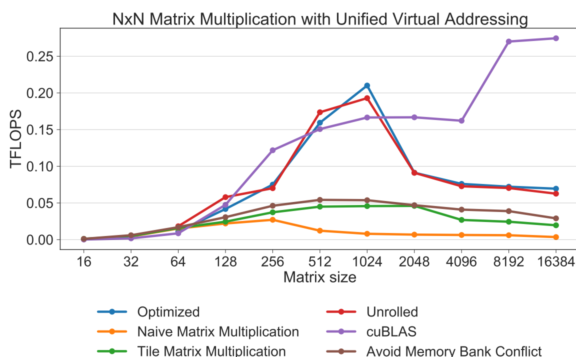

Fig. 6. 6 implementations of matrix multiplication on a NVIDIA P100 GPU for various matrix size using unified virtual addressing. All points are the average of 10 runs and the matrices are read from device memory

As we can see in Figure 4, this was indeed the case. The P100 can achieve a maximum of 9.3 TFLOPs, which the optimize CuBLAS version approaches, achieving a maximum of 8.95 TFLOPs at a matrix size of 16384. Our most optimized version achieves 4.07 TFLOPs, which is just under 2× slower than the CuBLAS version. While the exact algorithm is proprietary, we suspect that a variety of a factors contributed to the performance difference, including further cache optimization, and hand-tuned assembly, to achieve multiple floating-point operations per cycle. As a sanity check, the NVIDIA P100 has 3584 CUDA cores clocked at 1.328 GHz, which gives 4.759 TFLOPs assuming one operation per cycle.

The naive version of the matrix multiply achieves a maximum of 0.5 TFLOPS at a matrix size of 512 and 0.27 TFLOPS at a matrix size of 16384. We see that the naive version is over 15× slower than our most optimized version at the largest matrix size.

5.2 Impacts of memory management

As is apparent from Figure 4 and Figure 5, we see that the maximum performance achieved by manual memory management and unified memory management is approximately the same. However, we note that there is a dip in performance at intermediate matrix sizes for all methods when using manual memory management or UVA. This is due to the size of the shared memory for each block of CUDA cores. The NVIDIA P100 has a 64KB shared memory between each block of 64 CUDA cores. Because we are using a block size of 16 for each set of 64 CUDA cores, we use all of the shared memory for a matrix size of 1024x1024 (16 * 1024 * (32 / 8) = 64KB), which is exactly when we see the drop in performance. Increasing the matrix size by a factor of 2, then requires us to make two trips to fetch data. In the case of UVA, this requires going back to host memory, which dominates the cost, so we see throughput drop by close to a factor of 2. In the manual memory management case, we only need to go to global memory on the GPU, so there isn't as much of a drop in performance. In contrast, unified memory management avoids this problem entirely. Because the access pattern is known at compile time, unified memory can proactively fetch data from the GPUs global memory and hide additional latency.

We also saw that UVA performed *significantly* worse than either manual memory management or unified memory management. This is expected: every memory access must go over PCI-E. Notably, even CuBLAS was over 30× slower under UVA than manual or unified memory management. Our most optimized version was over 50× slower under UVA.

6 CONCLUSION

In this work, we analyze performance considerations when utilizing massively parallel hardware accelerators in the context of GPUs and matrix multiply. We evaluate 6 implementations of matrix multiply for various matrix sizes and show that optimizing the memory access patterns can lead to over a 15× speedup over the most naive algorithm. Similarly, we compare three CUDA memory management abstraction and see that *how* memory is accessed from the accelerator can cause a similarly dramatic effect, causing over a 40× slowdown.

REFERENCES

2017. The Top 500 Supercomputer List. (2017). https://www.top500.org/

Microsoft Research Blog. 2017. Microsoft unveils Project Brainwave for real-time AI. https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/. (2017). Accessed: 2017-09-04.

Ben Bours. 2017. GOOGLE AND MICROSOFT CAN USE AI TO EXTRACT MANY MORE AD DOLLARS FROM OUR CLICKS. (2017).

Lieven Eeckhout. 2017. Is MooreâĂŹs Law Slowing Down? WhatâĂŹs Next? *IEEE Micro* 37, 4 (2017), 4–5.

Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. 2004. GPU cluster for high performance computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*. IEEE, 47–47.

Graphcore. 2017. Accelerating Next Generation Machine Intelligence. https://www.graphcore.ai/technology. (2017). Accessed: 2017-09-04.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

Jianyu Huang, Tyler M Smith, Greg M Henry, and Robert A van de Geijn. 2016. Strassen's algorithm reloaded. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 690–701.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

NVIDIA. 2016. NVIDIA Tesla P100 Whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. (2016).

NVIDIA. 2017. NVIDIA Tesla V100 GPU Accelerator. https://images.nvidia.com/content/technologies/volta/pdf/437317-Volta-V100-DS-NV-US-WEB.PDF. (2017).

Tim C Schroeder. 2011. Peer-to-peer and unified virtual addressing.

Volker Strassen. 1969. Gaussian elimination is not optimal. *Numerische mathematik* 13, 4 (1969), 354–356.

Mark Wijtvliet. [n. d.]. Matrix multiplication in CUDA. ([n. d.]). http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda