Minimizing Data Transfers on Matrix Multiplications for GPU-based Heterogeneous Environments

Ricardo I. A. e Silva Jacques D. Brancher

State University of Londrina, Computing Department, Brazil

Abstract

Heterogeneous environments in computation systems present opportunity to reach teraFLOPS (TFLOPS)-level performance in common and useful operations such as matrix multiplications. But for that, tasks and its dataset must be parallelized and distributed. And more importantly, data transfers between processors memory can hinder global performance because they are slow. So a naïve implementation can fall short of performance with processors heterogeneity because of the required high amount of memory transfers. In this work we present a method of distributing tasks and dataset from matrix multiplications that minimizes memory transfers, while trying to utilize 2 GPUs and the CPU. We show that we minimize time spent with transfers from CPU to GPU down to 60% in comparison to execution in single GPU, for very large matrices cases.

Keywords: matrix multiplication, heterogeneous, GPGPU, parallelism, memory transfer, gemm

Authors' contact:
ricardoinacio@me.com
jacques@uel.br

1. Introduction

Since GPUs became programmable, through the incorporation of shader cores, there have been studies on reusing it for other tasks. It is called General Purpose computing on GPU (GPGPU). Because of its highly parallel nature some scientifically important algorithms presented promising results.

First implementations were done by hacking the existing programming model. Programmers had to map relevant data onto a texture memory object and design the algorithm as a shader program. Because it was unnatural, specific languages and tools for programming GPU were developed. Some of these are Cg, Brook+, CUDA, and OpenCL, with the two latters taking place as the standard.

As of late, increases of performance in CPU with each new generation have been lower than expected. New approaches are being tried, such as increasing number of CPU cores and integrating accelerators, such as FPGA and GPU. Studies show possibilities for utilizing GPGPU as if a specific core of the CPU, concurrently to its own cores. A system with this architecture is regarded as a heterogeneous environment, because it relies on different cores to run the same task.

Matrix multiplication is an important operation of linear algebra, because it is present in most applications of the latter. Also, it is very compute intensive. As such, an algorithm that implements it is key in achieving performance in others that are its dependents. The most common implementation is delivered by the BLAS library, and had its GEMM interface to be the standard. It is implemented by most other scientifically relevant linear algebra packages, such as MKL, ATLAS, GotoBLAS, FLAME and ACML.

In this work, we present a new approach to distributing tasks extracted from matrix multiplication between available processors in a 2-GPU 1-CPU heterogeneous environment. To check the method we implemented a well-known divide and conquer matrix multiplication.

2. Related Work

Matrix multiplications for CPU architectures are reviewed in Goto and van de Geijn [2008]. It explains that matrices should fit into L2 cache and avoid expensive TLB misses. Rünger and Schwind [2010] and Quintana-Ortí et al. [2009] addresses implementations on threaded and multi-core architectures. Because there are differences between processors and each new processor generation brings needs to optimize again, D'Alberto and Nicolau [2009] and ATLAS library take on automatically tuned routines. They are all relevant in extracting most FLOPS from CPU, which are still core element in heterogeneous environments.

The other important element, GPGPU, is profiled for linear algebra in Volkov and Demmel [2008]. Cui et al. [2010], Sun and Tong [2010], Tan et al. [2011] and Nakasato [2011] all work on coding optimizations for matrix multiplication kernels for various GPUs, from which computation reached TFLOPS performance ceiling in a single processor. Cui et al. [2010], Bosilca et al. [2011] and Du et al. [2011], the latter through MAGMA library, worked on portability of kernel performance between distinct GPU architectures, both from different generations and vendors. Lastly, Allada et al. [2009] showed relevancy of memory transfers and trustworthiness of OpenCL profiling tools on that subject.

On the actual field of matrix multiplication on heterogeneous environments with GPGPU, Ohshima et al. [2007] developed a method of parallelize tasks based on performance of each processor, so that both spend the same time computing. It works well on some cases, but it does not minimize memory transfers nor the amount of data to be transferred. Igual et al. [2011] developed a run-time scheduler capable of decomposing linear algebra operations, parallelize tasks, executing out-of-order based on dependencies and is GPGPU aware. It delegates resolution of matrix multiplication to specialized BLAS libraries.

3. Parallelization on Heterogeneous Environments

Common heterogeneous environments are built around a CPU, added by discrete GPUs, through a high speed interconnection bus. In such cases, each processor has its own memory space. In this kind of system it is possible to execute algorithms that involve every processor. To achieve that, data has to be modeled so it can be explicitly divided between memory spaces. More importantly, programmers have to carefully consider memory transfers when designing algorithms.

On current systems, memory transfers between CPU and GPUs usually are the bottleneck in many algorithms implementations. GPUs already reached few teras of float-point operations per second (TFLOPS) in theoretical peak of computational power, and memory bandwidth of ~300GB/s in its own memory space. At the same time, the best CPU–GPU interconnection implementation, the PCIe 3.0, has theoretical bandwidth peak of only 16GB/s. For this reason, to better take advantage of GPGPU potential in heterogeneous environments, memory transfers must be minimized.

4. Matrix Multiplication

There are various matrix multiplication algorithms, each with its features and pitfalls. For instance, naïve algorithm is small and straightforward, but has asymptotic cost of O(n^3) and has a CPU-unfriendly memory access pattern. On the other hand, the elaborate Strassen algorithm features asymptotic cost of O($n^{2.802}$), but adds considerable overhead in terms of floating-point operations (flops) and memory access (memops) and usage.

An interesting alternative is the divide-and-conquer variation of the naïve algorithm. It still has asymptotic cost of O(n^3) and, like in Strassen, introduces overhead in terms of flops and memops, albeit smaller. But it introduces the ability to parallelize and redefine the size of dataset being operated. Also, as we show, there is the possibility of reducing the amount of data to be transferred to each processor involved in the algorithm execution.

4.1 Blocking Scheme and Tasks Definition

Consider a matrix multiplication given by C = AB, where $A \in R^{m \times k}$, $B \in R^{k \times n}$ and $C = R^{m \times n}$, with m, n and k $\in N^*$. Matrices are divided so that

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix},$$

with each $A_i \in R^{\frac{m}{2} \times \frac{k}{2}}$, $B_i \in R^{\frac{k}{2} \times \frac{n}{2}}$ and $C_i \in R^{\frac{m}{2} \times \frac{n}{2}}$. From there, we proceed with naïve matrix multiplication to extract the following set of equations

$$\begin{aligned} C_1 &= A_1B_1 + A_2B_3, \\ C_2 &= A_1B_2 + A_2B_4, \\ C_3 &= A_3B_1 + A_4B_3, \\ C_4 &= A_3B_2 + A_4B_4. \end{aligned}$$

Then, we choose to represent C_i elements of matrix C in terms smaller matrix multiplications between A_i and B_i, so that

$$\begin{aligned} C_1 &= M_1 + M_2, \\ C_2 &= M_3 + M_4, \\ C_3 &= M_5 + M_6, \\ C_4 &= M_7 + M_8. \end{aligned}$$

From that, we are able to define a set $M = \{M_1, \dots, M_8\}$. These are the smaller matrix multiplications needed to resolve the original. They are given by

$$\begin{aligned} M_1 = A_1B_1,\ M_2 = A_2B_3,\ M_3 = A_1B_2,\ M_4 = A_2B_4, \\ M_5 = A_3B_1,\ M_6 = A_4B_3,\ M_7 = A_3B_2,\ M_8 = A_4B_4. \end{aligned}$$

So, we segment operations involved in C = AB in two classes: matrix multiplications and matrix additions.

4.2 Execution

To avoid unnecessary memory transfers, we maximize memory reuse. So, if a matrix block is sent to a device, it should run all tasks that the specific matrix block is involved. And, because this would be impossible on a regular matrix multiplication, we choose to ignore matrix additions.

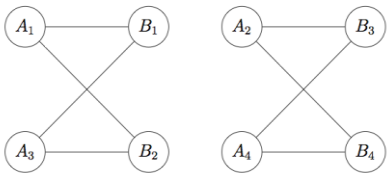

Figure 1: Relationship graph between matrices blocks that are factors of a same multiplication.

With this in mind, we model a graph G = (V, E) representing the relationship between matrices blocks, where vertices set V are matrices blocks and edges set E represents participation as factors in a multiplication.

Such graph is depicted in figure 1. From there, it is clear that there are two sets of independent blocks.

Because multiplications only have two factors, edges E represent not only the relationship of factor blocks, but a multiplication. So we can clarify even further by defining multiplications graph $G_m = (V_m, E_m)$, where vertices $V_m = E$ and edges $E_m = V$, depicted in figure 2. It shows which multiplications should be executed on the same processor.

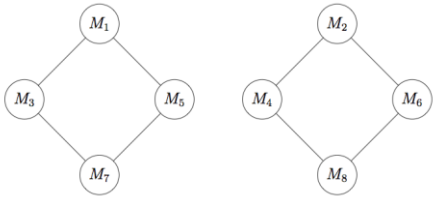

Figure 2: Multiplications graph depicting multiplication operations that depend on a same factor.

With this procedure, there are three groups of tasks, as shown by figure 3. Two are independent set of multiplications and one is a set of matrix additions that depends on results of the multiplications. What we gather from it is that multiplications should be executed by accelerator devices. When finished, they return results to CPU that executes the more affordable additions, thus reassembling the resulting matrix.

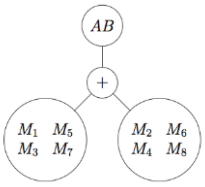

Figure 3: Tree representing extracted tasks in naïve blocking matrix multiplication.

5. Experiments

For experimentation to the proposed method, we implemented an algorithm, called 'hybrid', which receives already parted matrices, distribute blocks to processors accordingly, send operation commands, receive results blocks and merge them in a single matrix by adding on CPU. For that, our implementation assumes usage of hierarchical matrices, which are already perfectly parted. For this reason, we avoid *memcpy()* that otherwise would be needed to format matrices.

Our implementation utilized clAmdBlas 1.8 SGEMM routine, with AMD APP SDK 2.7, when executing multiplication on GPU. To comply with the SDK, communications with GPU were done through OpenCL calls. We also implemented a naïve CPU threaded matrix addition, through OpenMP framework, for use in merging results coming from GPU.

Since we did not readily have two equal GPUs, we assembled our system with two distinct, the other being an Radeon HD 4830. While it is not capable of running our hybrid method, because it is not supported by clAmdBlas, we did test the interference of both GPU's on each other at data transfers between CPU and GPU. We found that interference compromised no more than 7% in data transfers performance, since both together could not reach PCIe 2.0 bandwidth peak. Also, because even two Radeon HD 6850 would not reach this peak, we assumed our system to have two of that same GPU. So we configured our method to execute only a group of multiplications.

Because we aimed at reducing data transfer time from CPU to GPU, we compare our method to a straight call to clAmdBlas' SGEMM. For that end, we utilized available profiling tools from OpenCL run-time platform. With them we profiled timing of stages of execution. Experiments ran on the system described in table 1.

Table 1. Specification of the system used in experiments

CPU	Core i5 760 2.8GHz
Memory	4GB DDR3 1333MHz
OS	Windows 7 (64 bits)
GPU	Radeon HD 6850
Graphics Bus	PCIe 2.0
VRAM	1GB
GPU processor clock	820MHz
GPU memory clock	1050MHz

5.1 Results

Timings gathered from experimentation are shown in table 2. Method column indicate method utilized in the following row. Size column indicate matrices sizes, the same for all dimensions (m = n = k). CPU to GPU column indicates time to transfer every input data from CPU memory space to GPU, in seconds. GPU kernel column indicate time for the GPU to finish calculations, in seconds. GPU to CPU column indicates time to transfer results back from GPU memory space. Lastly, CPU addition column indicates time spent in merging results by adding matrices received from GPU, in seconds. Obviously, since this stage is only present in hybrid method, only its rows have results for that column. Data is also displayed on figure 4, for better visualization.

Table 2. Experiments timings.

Method	Size	CPU to GPU (s)	GPU kernel (s)	GPU to CPU (s)	CPU addition (s)
Hybrid	2000	0.0079	0.0144	0.0071	0.0070
	4000	0.0224	0.0798	0.0203	0.0277
GPU-only	2000	0.0113	0.0199	0.0048	
	4000	0.0371	0.1544	0.0172	

5.2 Discussion

From figure 4 we see that currently our method only truly benefits when multiplying large matrices. We confirmed the efficiency of our methodology about memory transfers from CPU to GPU for cases of matrices sizes greater than 2000x2000. At that size, the said transfer for hybrid method is about 70% of the time of GPU-only. At 4000 sizes, hybrid is about 60% of GPU-only.

Also, we note the presence of overhead in both kinds of transfers in hybrid method compared to GPU-only. This is because the latter only issue one copy command on a single block of dataset on memory, while the former must issue four copy commands.

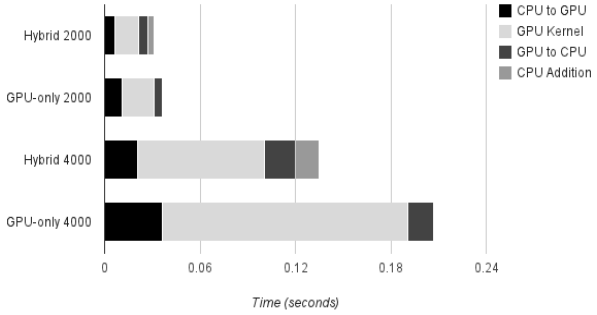

Figure 4: SGEMM 'hybrid' and 'GPU-only' profiles comparison.

6. Conclusion

In this work, we present a new approach to distributing tasks extracted from matrix multiplication between available processors in a 2-GPU 1-CPU heterogeneous environment so that each GPU receives the minimum necessary dataset for operating in parallel with the other. Through the results of our implementation, we prove the efficiency of that method for large matrices. Starting from 2000×2000 matrix multiplications, time for transfers from CPU to GPU are reduced by 30%. And it can reduce down to 60%, in 4000×4000 matrices size case.

Now we set to investigate if that gain still holds true in cases where usage of two GPUs saturates available PCIe bandwidth. Also, we should investigate if rectangular-shaped memory transfers from CPU to GPU do introduce significant overhead, as is available in OpenCL calls. Since it still is unusual for matrices on memory to be perfectly hierarchy-parted, this would be a wanted feature, if viable.

References

Allada, V. et al. 2009. Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster. *In Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–9.

Bosilca, G. et al, 2011. Performance Portability of a GPU Enabled Factorization with the DAGuE Framework. *In Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pp. 395–402.

Cui, X. et al 2010. Auto-tuning Dense Matrix Multiplication for GPGPU with Cache. *In Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems, ICPADS '10*, pp.237–242, Washington, DC, USA.

D'Alberto, P. and Nicolau, A., 2009. Adaptive Winograd's matrix multiplications. *ACM Trans. Math. Softw.*

Du, P. et al, 2011. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*.

Goto, K. and van de Geijn, R. A., 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*

Igual, F. D. et al, 2011. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing*, 2011.

Nakasato, N., 2011. A fast GEMM implementation on the cypress GPU. *SIGMETRICS Perform. Eval. Rev.*

Ohshima, S. et al, 2007. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. *In Proceedings of the 7th international conference on High performance computing for computational science, VECPAR'06*, pp.305–318, Berlin, Heidelberg.

Quintana-Ortí, G. et al, 2009. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*

Rünger, G. and Schwind, M., 2010. Fast recursive matrix multiplication for multicore architectures. *Procedia Computer Science*.

Sun, Y. and Tong, Y., 2010. CUDA Based Fast Implementation of Very Large Matrix Computation. *In Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2010 International Conference on*, pp.487–491.

Tan, G. et al, 2011. Fast implementation of DGEMM on Fermi GPU. *In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, pages 35:1-35:11, New York, NY, USA.

Volkov, V. and Demmel, J. W., 2008. Benchmarking GPUs to tune dense linear algebra. *In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08*, pp.31:1-31:11, Piscataway, NJ, USA.