Thesis Synopsis – Aditya Deshpande

Combining Data Parallelism and Task Parallelism for Efficient Performance on Hybrid CPU and GPU Systems

In earlier times, computer systems had only a single core or processor. The number of transistors on-chip (i.e. on the processor) doubled every two years. Intelligent use of these transistors by computer architects led to faster hardware. The clock speed of these processors kept improving. As a result, all applications enjoyed free speedup (popularly known as *free ride*) every few years. This phenomena was termed as the Moores Law. With more and more transistors being packed on-chip, power consumption became an issue, frequency scaling reached its limits and industry leaders eventually adopted the paradigm of multi-core processors. Computing platforms of today have multiple cores and are parallel. CPUs have multiple identical cores. A GPU with dozens to hundreds of simpler cores is present on many systems. In future, other multiple core accelerators may also be used.

With the advent of multiple core processors, the responsibility of extracting high performance from these parallel platforms shifted from computer architects to application developers and parallel algorithmists. Tuned parallel implementations of several mathematical operations, algorithms on graphs or matrices on multi-core CPUs and on many-core accelerators like the GPU and CellBE, and their combinations were developed. Researchers exploited the coarse parallelism offered by multiple cores of the CPU and used the cache memory for accelerating several applications. Intel Math Kernel Library was developed as a suite of standard operations viz. fast fourier transform, LU, Cholesky and QR decomposition, random number and probability distribution generator, splines and interpolation etc. that made the best use of multi-core CPU to provide optimum performance. Parallel algorithms developed for multi-core CPUs primarily focussed on decomposing the problem into a few independent chunks and using the cache efficiently.

As an alternative to CPUs, Graphics Processing Units (GPUs) were the other most cost-effective and massively parallel platforms, that were widely available. Researchers have thus always been interested in harnessing their power for general purpose computation. Prior to development of frameworks such as CUDA or OpenCL, people programmed the shaders in a graphics pipeline intelligently to perform general purpose computations. With CUDA it became easier to express data parallelism and exploit the computation resources on the graphics processor for general purpose computations. Frequently used algorithmic

primitives such as sort [11, 2, 6, 5], scan [11], sparse matrix vector multiplication [1], graph traversals [7] among others were efficiently implemented on GPU using CUDA. A suite of statistical, data structures, arithemtic, image, signal processing related primitives for the GPU, similar to Intel MKL, was developed and distributed with NvPP [9] and other libraries viz. Thurst [8], CUDPP [4]. These parallel algorithms on the GPU decomposed the problem into a sequence of many independent steps operating on different data elements. These steps could be performed on multiple simple cores of the GPU and they used the shared memory (for every small group of threads) effectively.

But the above operations – statistical, or on graphs, matrices and list etc. – constitute only portions of an *end-to-end* application and in most cases these operations also provide some inherent parallelism (task or data parallelism). Matrix operations like QR decomposition, matrix multiplication or sparse matrix vector multiplication involved performing same operations across different data elements of the matrix. Though the nodes vary at every step, many graph traversals involved performing the same operations on different nodes without any dependence between the nodes at every step. Many image processing operations such as filtering, color conversion, fast fourier transform were by its very nature data parallel, i.e. operations on all pixels were independent of each other. This led to these operations being amongst the ones adapted early to all available parallel platforms. But, the problems which lack such task or data parallelism are still difficult to map to any parallel platform, either CPU or GPU. In this thesis, we consider a few such difficult problems – like Floyd-Steinberg Dithering and String Sorting – that do not have trivial data parallelism and exhibit strong *sequential dependence* or *irregularity*. We show that with appropriate design principles we can find data parallelism or fine-grained parallelism even for these tough problems. Our techniques to break sequentiality and addressing irregularity can be extended to solve other difficult data parallel problems in the future.

The Error Diffusion Dithering application, specifically Floyd-Steinberg Dithering (FSD) algorithm, exhibits sequentiality. In this thesis, we develop parallel algorithm for FSD and develop techniques to break sequentiality found in many problems. We note that, though error distribution scheme of FSD implies a dependency of even the last pixel (bottom right of image) on the first (top left), our analysis allows us to find pixels that can be processed independently. This error distribution scheme imposes a constraint that pixels separated by a knight's move can be processed independently. We start with a wavefront at top left of the image and process a knight's move pixel boundary per iteration in parallel. We perform these parallel processing steps on the independent pixels till we reach the bottom right of image (and achieve the final output). Our data parallel approach achieves a speedup of $10\times$ on high-end GPUs and a speedup of about $3-4\times$ on low-end GPUs. The pattern of data dependency that is found in FSD is commonly observed in many dynamic programming problems and our techniques could be used to solve those.

The problem of String Sorting involves sorting long and mostly variable length keys. Since any two given keys match to different lengths, the work performed by any two threads is not uniform. Also, arbitrary memory accesses need to be performed depending on the ordering between the input keys. Algorithms where work given to different threads is different and the memory accesses are arbitrary depending on the input, are popularly known as irregular algorithms. Thus, string sorting exhibits the characteristics of an irregular algorithm. As part of this thesis we develop a fast parallel algorithm for this irregular problem of string sorting. The approach we develop involves mapping the operations of string sort to fast standard primitives of fixed-length sort, scatter and scan. Highly tuned implementations are available on the GPUs for all these standard primitives and their use provides high performance in any application. The challenges of irregularity are efficiently handled within these primitives. We leverage the intelligent design of primitives and map the original irregular problem to a sequence of steps involving only standard primitives (which have an efficient solution). This primitive-based approach will allow for fast development of parallel and high performing data-parallel applications for many other irregular problems. Also, any improvements to the primitives will be directly inherited by these applications without requiring re-design. Our string sort approach achieves a speed up of around $10 - 19\times$ as compared to state-of-the-art GPU merge sort based methods [8, 3] on difficult datasets. We extend our string sorting algorithm to efficiently solve another irregular problem of Burrows Wheeler Transform. Our Burrows Wheeler Transform implmentation is the first to achieve speed up on the GPU.

It is not enough to have a truly fine-grained parallel alogrithm for only a few operations. Any end-to-end application consists of many operations, some of which are difficult to execute on a fine-grained parallel platform like GPU. At the same time, computing platforms consist of CPU and a GPU which have complementary attributes. CPUs are optimized for running a sequential code, most of the hardware (or transistors) are dedicated to finding intruction level parallelism, perform branch prediction etc. CPUs are suitable for some heavy processing by only a few threads i.e. they prefer task parallelism. The sequential portions of the application should thus be performed on CPUs. In GPUs, most transisitors are used for more direct problem sovling purposes. Thus GPUs offer high performance benefits using all these transistors for data parallel operations. Applications can achieve optimal performance by combining data parallelism on GPU with task parallelism on CPU. In this thesis, we examine two methods of combining data parallelism and task parallelism on a hybrid CPU and GPU computer system: (*i*) *pipelining* and (*ii*) *work sharing*.

We study the Burrows Wheeler Compression (BWC) implementation in Bzip2 and show that best performance can be achieved by pipelining its different stages effectively. For compute intensive tasks (i.e. BWT) we develop a data parallel algorithm on GPU and we perform other tasks (merge, MTF, Huff-

man encoding) on CPU. We efficiently overlap the CPU and GPU computations and ensure that resources do not remain idle. In contrast, a previous GPU implementation of BWC by Patel et al. [10] performed all the tasks (BWT, MTF and Huffman encoding) on the GPU. Each of the task was slower on the GPU and their resulting implementation was about 2.78× slower than CPU. Using our pipelining strategies we develop a hybrid BWC and achieve a 2× speedup over CPU on the same problem. This hybrid BWC approach uses only a single CPU core, whereas the CPU still has more cores. To use them effectively, we perform BWC tasks on a few blocks (using best sequential implementation of BWC) on idle CPU cores in parallel with our hybrid BWC. We call this the all-core BWC, which uses all cores of the CPU as well as the many core GPU. We improve nearly 2× speedup using hybrid BWC to 4.87× with our all-core BWC on a high-end platform.

For the problem of FSD, as discussed earlier, we first develop a data parallel algorithm using data-dependency to our advantage. We observe that the parallelism is low towards the start and end for this data parallel algorithm. We solve these low parallelism portions on the CPU and GPU solves only the high parallelism portions. This is called the Handover FSD algorithm. We further extend this to the Hybrid FSD algorithm. In Hybrid FSD algorithm, the CPU does a portion of the work even when GPU is operating in the high parallelism region. This Hybrid algorithm involves concurrent processing by both CPU and GPU. Though a transfer of few bytes of memory is required per iteration to keep the CPU and GPU synchronized, we show that the overall performance improves even with the memory transfer overhead. Since we split the same data parallel FSD step across CPU and GPU, we call this work sharing instead of pipelining. Today, there is an increased interest in unifying the virtual memory address space for CPU and GPU. This unification will make memory transfer between CPU and GPU even faster making work sharing data parallel algorithms necessary for high performance.

In conclusion, we develop data parallel algorithms on the GPU for difficult problems of Floyd-Steinberg Dithering, String Sorting and Burrows Wheeler Transform. In earlier literature, simpler problems which provided some degree of data parallelism were adapted to the GPUs. The problems we solve on GPU involve challenging sequential dependency and/or irregularity. We show that in addition to developing fast data parallel algorithms on GPU, application developers should also use the CPU to execute tasks in parallel with GPU. This allows an application to fully utilize all resources of an end-user's system and provides them with maximum performance.

References

[1] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[2] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. J. Exp. Algorithmics, 14:4:1.4–4:1.24, January 2010.

[3] Andrew Davidson, David Tarjan, Michael Garland, and John D. Owens. Efficient parallel merge sort for fixed and variable length keys. In Proceedings of Innovative Parallel Computing (InPar '12), May 2012.

[4] CUDPP Developers. Cudpp, 2013. Version 2.1.

[5] Alexander Gress and Gabriel Zachmann. Gpu-abisort: Optimal parallel sorting on stream architectures. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium, page 45, 2006.

[6] Linh K. Ha, Jens Krüger, and Claudio T. Silva. Fast four-way parallel radix sorting on gpus. Computer Graphics Forum, 28:2368–2378, 2009.

[7] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In Proceedings of the 14th International Conference on High Performance Computing, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] Jared Hoberock and Nathan Bell. Thrust a parallel template library, 2010.

[9] Nvidia. Nvidia peformance primitives.

[10] Ritesh A. Patel, Yao Zhang, Jason Mak, and John D. Owens. Parallel lossless data compression on the GPU. In Proceedings of Innovative Parallel Computing (InPar '12), May 2012.

[11] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.