Algorithm Engineering Challenges in Multicore and Manycore Systems

Algorithm Engineering Herausforderungen bei Mehrkern- und Manycore-Systemen

Seunghwa Kang, David Ediger, David A. Bader, Georgia Institute of Technology, Atlanta, USA

Summary Modern multicore and manycore systems have the strong potential to deliver both high performance and high power efficiency. The large variance in memory access latency, resource sharing, and the heterogeneity of processor architectures in modern multicore and manycore systems raise significant algorithm engineering challenges. In this article, we overview important algorithm engineering issues for modern multicore and manycore systems, and we present algorithm engineering techniques to address such problems as a guideline for practitioners. ►►►

Zusammenfassung Moderne Mehrkernsysteme und Manycore Prozessoren erlauben nicht nur hohe Performance sondern auch geringen Energieverbrauch. Die große Varianz bei Speicherzugriffszeiten, die Möglichkeit gemeinsame Ressourcen zu nutzen und die Heterogenität der Prozessorarchitekturen in solchen Systemen erzeugen erhebliche Herausforderungen für den Bereich des Algorithm Engineering. In diesem Artikel geben wir eine Übersicht der wichtigsten Belange des Algorithm Engineering für Mehrkern- und Manycore-Systeme und stellen Methoden des Algorithm Engineerings vor, die dem Praktiker als Leitlinien bei derartigen Problemen dienen können.

Keywords D.1.3 [Software: Programming Techniques: Parallel Programming]; performance tuning, GPU ►►►
Schlagwörter Parallele Programmierung, Algorithmen

1 Introduction

Multicore processors are replacing single-core processors from smartphones to supercomputers. Manycore accelerators, such as graphics processing units (GPUs), are also becoming popular. Multicore processors and manycore accelerators have the potential to deliver both high performance and high power efficiency [1; 11]. However, to realize such potential, software must be parallelized and tuned for these processors. This is the task for today's algorithm engineers. Algorithm engineering refers to the process of transforming an algorithm on paper into a robust and efficient implementation on today's computers [3]. There have been significant efforts to automate this parallelization and tuning process (e. g., [4; 12]), but the problem remains a daunting challenge. Algorithm engineering work for modern systems will largely remain the programmer's burden for the foreseeable future.

Parallelizing an algorithm, obtaining a good load balance across processors, and attaining a speed-up are common challenges when using parallel machines; parallelization becomes more challenging for parallel systems with multi-core processors and accelerators than parallel systems with single-core processors as the requirement for parallelism increases. In addition to these, algorithm engineering for multicore and manycore processors is challenging for three major reasons: the large variance in memory access latency, the sharing of system resources, and heterogeneity. Even in a single-core system, memory access latency to the closest memory (registers) is much smaller than the latency to the farthest memory (e. g., DRAM). The variance becomes larger in computers with multicore processors and manycore accelerators. For a system with a traditional microprocessor (CPU) and an additional GPU card, accessing the GPU's memory from the CPU side takes much longer than accessing the CPU's memory. In multicore systems, multiple cores often share resources such as a cache and memory bandwidth. Multithreaded microprocessors can spawn multiple hardware

threads on a single core, and in this case, the hardware threads also share resources within a core. In each of these cases, the microprocessor may have different architectural features. For example, CPUs and GPUs both support vector instructions, which apply a single instruction to multiple scalar values, but the vector units of a CPU and a GPU have significantly different structures. The CPU's vector unit supports packing of sixteen 8-bit operands in a single 128-bit execution unit while today's (NVIDIA) GPUs can process only one 8-bit operation per 32-bit wide vector lane (an NVIDIA Fermi GPU's vector unit has sixteen vector lanes).

All the above factors affect the performance of a program in different ways for different target systems. An algorithm engineering approach that works well for one system often does not work for another system or input data set. To achieve high performance, programmers need to understand the characteristics of their target system, as well as their data, and properly tune their code for both. This article aims to help programmers with the former. We will give an overview of architectural characteristics of modern multicore and manycore systems and present algorithm engineering techniques for these systems.

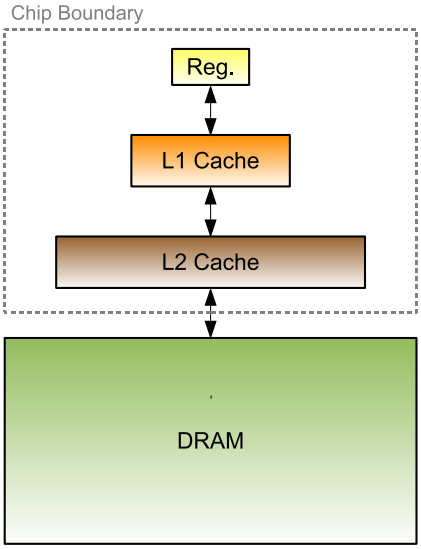


Figure 1 Illustration of the classical single-core memory hierarchy.

2 Variance in Memory Access Latency

Typical modern computer systems, including single-core systems, have a hierarchical memory subsystem. Assume that a memory closer to an arithmetic logic unit (ALU) resides in a higher position of the memory hierarchy. Then the highest level memory (registers) is the smallest in size but has the lowest latency and highest bandwidth. As shown in Fig. 1, a typical single-core system has level 1 instruction and data caches, a level 2 cache, and a DRAM in the lowest level. A lower level memory is larger but has higher latency and lower bandwidth.

Computers with a multicore CPU have a more complex memory subsystem. Data can reside in another core's cache that often has higher latency than the lower level cache shared by the cores. In computers with multiple multicore CPUs, data can reside in the cache of another core of the same CPU or a cache of another CPU in a different socket. The variation in DRAM access latency also increases. Recent Intel and AMD CPUs adopt the NUMA (Non-Uniform Memory Access) architecture. In the NUMA architecture, each CPU has its own local DRAM. The latency to the local DRAM is lower than the latency to a remote DRAM (another CPU's local DRAM). In a CPU-GPU hybrid system, accessing the GPU's DRAM from the CPU side incurs even higher latency.

Figure 2 illustrates the memory hierarchy of a CPU-GPU hybrid system with two CPUs and one GPU.

To access data in the same CPU, only on-chip communication is required; on-chip communication typically incurs much lower latency than off-chip communication.

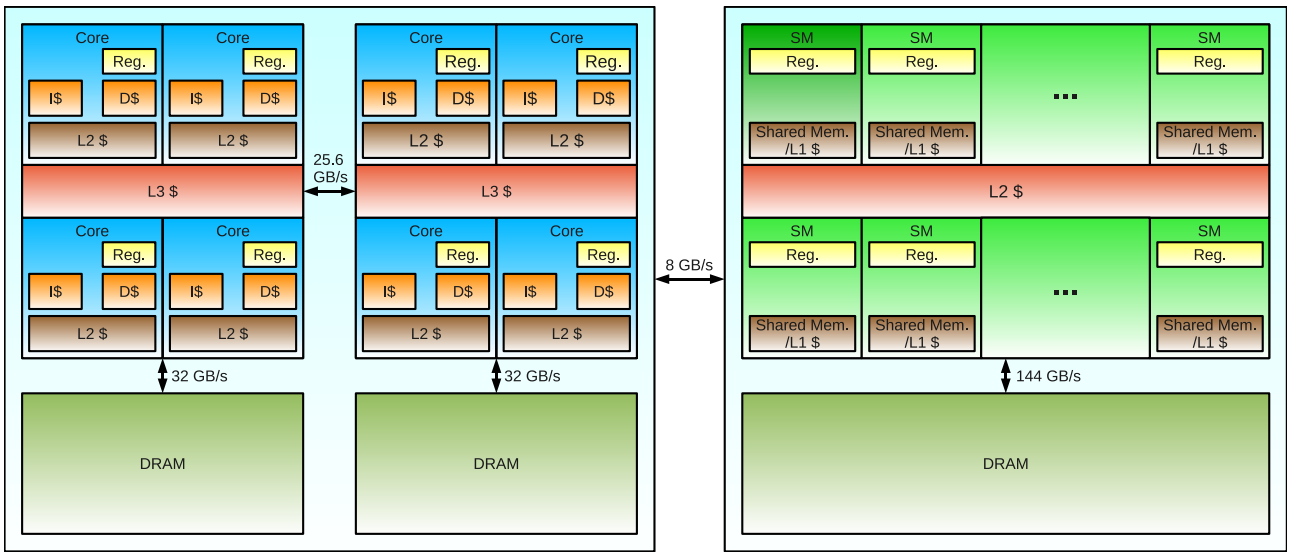


Figure 2 Illustration of the Memory Hierarchy of a CPU-GPU Hybrid System (Two Intel X5550 CPUs and an NVIDIA C2050 GPU).

In contrast, to access the GPU's DRAM, off-board communication between the mainboard and the GPU board across the PCI Express bus is required. This incurs much higher latency and often becomes a performance bottleneck [8].

2.1 Temporal and Spatial Locality

Due to the increase in the variance of memory access latency, it is much more important for multicore and manycore systems to minimize the impact of long-latency, low-level memory accesses. One way to achieve this is to increase the temporal and spatial locality of data accesses. Temporal locality refers to the degree of data reuse in a short time duration. If data are accessed repeatedly in a short time duration, the data have a high probability of residing in a small but low latency memory (such as registers and level 1 cache), except during the first access. Say we need to apply operations a, b, and c to every element of matrix A. One way to implement this is to apply operation a to every element of matrix A first, then apply operation b, and finally operation c to every element of matrix A. In this case, there is a significant time gap between the accesses of a matrix element. The second approach is to apply operations a, b, and c to a matrix element before applying operations to the next element. This approach has much higher temporal locality.

The well known blocking technique also improves the temporal locality of data accesses. Say we are multiplying m by n matrix A and n by l matrix B to compute m by l matrix C using the elementary matrix-matrix multiply algorithm. We can compute the matrix C's ith row and jth column element (C_{ij}) as the following.

$$C_{ij} \leftarrow \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, 1 \le i \le m \text{ and } 1 \le j \le l$$

Figure 3 depicts the data access patterns with (bottom) and without (top) blocking. Non-blocking matrix multiplication code computes C_{ij} by accessing the ith row of matrix A and the jth column of matrix B. If the code updates C_{11} first and C_{12} second, A_{11} is accessed first to update C_{11} and is accessed again to update C_{12} after all the elements in the first row of matrix A are accessed. The gap between two consecutive accesses is n elements. The blocking approach partitions matrix A, B, and C to small blocks. If we consider a block as a single matrix element, blocking matrix multiplication code accesses the blocks in the same way to the non-blocking code accesses the matrix elements. Also, when multiplying two blocks (one block from matrix A and one block from matrix B) to update a block in matrix C, the code accesses the elements in each block in the same way as the non-blocking code. In this case, assuming that the block size is b by b, the first element in the first row of matrix A's block is accessed once per every b element accesses. The gap between two consecutive accesses is b instead of n. This significantly improves temporal locality if n is much larger than b.

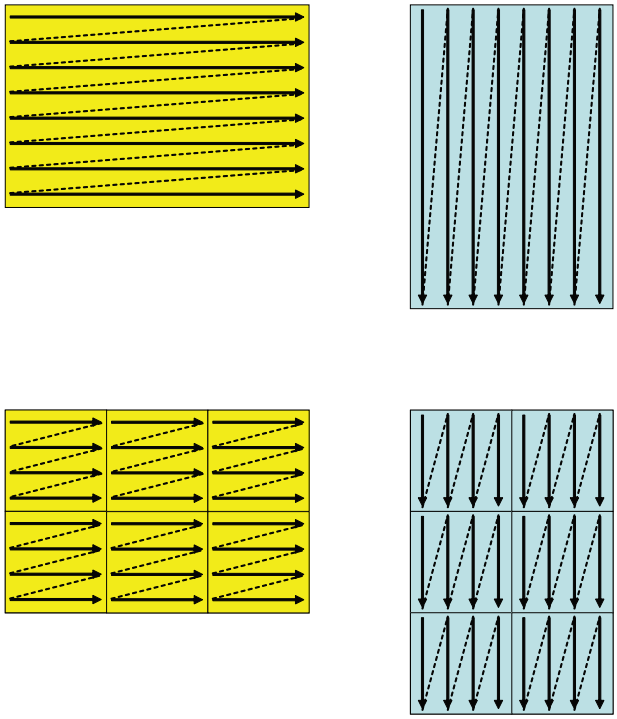


Figure 3 Data access patterns without (top) and with (bottom) blocking. Assume the matrices are stored in row major order.

High temporal locality is even more important for CPU-GPU hybrid systems. Once data are loaded to the GPU side, GPUs often can process the data very quickly. However, moving data between the CPU side and the

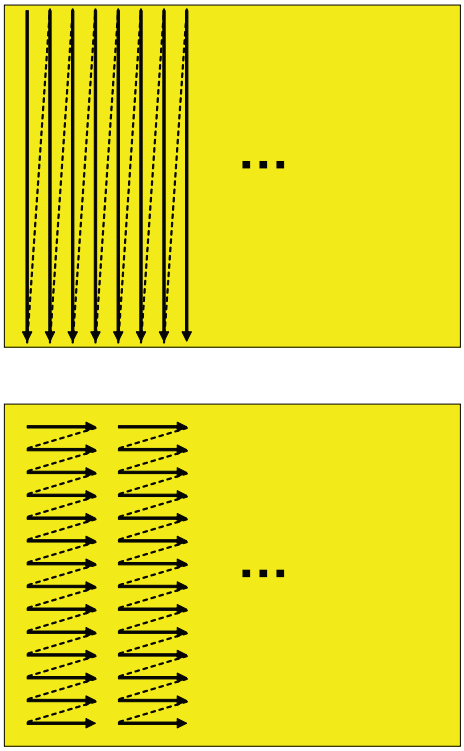


Figure 4 Data access patterns without (top) and with (bottom) column grouping. Assume the matrix is stored in row major order.

GPU side across the PCI Express bus incurs high latency. Algorithm engineering to achieve high temporal locality becomes even more important in CPU-GPU hybrid systems due to the high off-loading overhead across the PCI Express bus and the GPU's high computing power.

Modern CPUs load and store data from DRAM to cache or from cache to DRAM in a cache line granularity (often 32 bytes, 64 bytes, or 128 bytes). If multiple data elements in the same cache line are accessed in a short time duration, then we can say the data accesses have high spatial locality. We can often increase spatial locality by reordering data accesses. Column grouping (e. g., [6]) is an example. Figure 4 illustrates column grouping.

A particular algorithm may require us to sum the elements of each column. The top figure in Fig. 4 depicts the data access pattern without column grouping. If the matrix is stored in row-major order, consecutive data accesses hit different cache lines. This data access pattern has low spatial locality and low performance. We can compute the same results by reordering data accesses as shown in the bottom figure of Fig. 4; we can partially update the sum of each column in a horizontal sweep of a cache line instead of updating the sum of one column after finishing the summation of the previous column. This modification, or column grouping, leads to much higher spatial locality.

2.2 Affinity

The affinity of computation and data is especially important for a system with multiple CPUs and the NUMA architecture. In a NUMA system with two CPUs, each CPU has its own local DRAM (such as the system depicted in Fig. 2). If a thread running on one CPU accesses data in its local DRAM, the thread can directly access the data using the local memory controller of the CPU that the thread is currently running on. If the thread accesses data in a remote DRAM, the thread needs to access the data via another CPU, and this further increases memory access latency. We can reduce the number of remote DRAM accesses by binding a thread and the thread's frequently accessed data to a CPU and its local DRAM. The NUMA API [9] provides mechanisms to control the binding to improve affinity.

It is often helpful to replicate read-only and heavily read data in a system with multiple CPUs and the NUMA architecture (e. g., [8]). If the data are significantly larger than the cache size, the data need to be read multiple times from DRAM. If the data are replicated – one copy per CPU and its local DRAM pair – threads running on a CPU can always read data from its local DRAM instead of a remote DRAM. Initial replication cost can be amortized if the data are read multiple times.

2.3 Latency Hiding

We can reduce a number of high latency memory accesses by increasing the locality of data accesses. However, in most practical cases, we cannot completely remove high latency memory accesses. This necessitates a technique to hide memory access latency. Two popular approaches to hide latency are *prefetching* and *multithreading*.

Prefetching works for cases with predictable data access patterns. As an example, we first process block *A* and then block *B*. With prior knowledge that we will access block *B* after processing block *A*, we can preload block *B* while processing block *A*. Hardware prefetching tracks data access patterns using hardware components. Hardware prefetching does not require explicit software control but works only for limited cases – e. g., a linear scanning of an array with stride one. In many cases, high level knowledge is necessary to predict future data accesses. In such cases, explicit software control is necessary to preload data while processing another data, or in other words, to overlap communication and computation.

Double buffering is a representative approach to overlap communication and computation. Double buffering maintains two buffers. While processing the data in one buffer, another buffer is filled. Implementing double buffering requires an explicit mechanism to control data transfer. Message Passing Interface (MPI) provides such a mechanism, although it is used most often in large compute clusters.

Prefetching is effective if future data access patterns are predictable. However, many irregular applications have unpredictable data access patterns even with high level knowledge. Graph algorithms fall into this category. Figure 5 provides an example.

A thread may access vertex *A*, then vertex *B*, and finally vertex *C* in the figure. However, in many cases, the thread does not know that it will access vertex *B* until it has finished processing vertex *A*. Similarly, the thread cannot predict that it will access vertex *C* before processing vertex *B*. Prefetching will not work for this type of problem. Multithreading is helpful in this case if we can extract sufficient parallelism to hide the latency of memory accesses. Several multicore processors support multiple hardware threads on a single core. For example, the Intel Nehalem architecture supports two

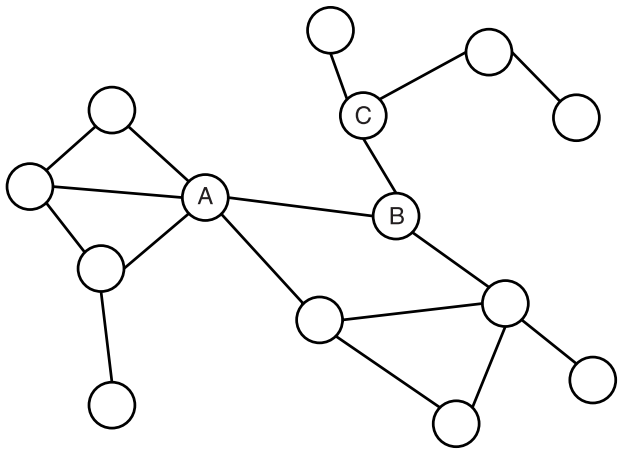


Figure 5 Unpredictable data access patterns in many graph algorithms.

threads per core. The Sun UltraSparc T1 and T2 support four and eight threads per core, respectively. The IBM Power7 supports four threads per core. The Cray XMT, which is specially designed to support applications with a large memory footprint and irregular data access patterns, has 128 threads per core. The hardware threads in a core share arithmetic logic units (ALUs) and other execution units in the core. If there is only one thread per core and the thread executes a large number of remote memory accesses, the thread will spend a large fraction of time waiting for remote memory accesses to finish and does not have other instructions for the ALUs and other execution units on the chip. The core suffers from severe underutilization. If there are multiple threads, an execution unit in the core can be utilized if at least one thread has an instruction ready for the unit. If we can implement an irregular application using multiple parallel threads, hardware multithreading can effectively hide the latency of distant memory accesses (see [2] for an example).

3 Resource Sharing

In single-core systems, multiple threads can share the resources of a core in a time-sharing fashion (often referred as multi-tasking), but there is no resource sharing among concurrently executing threads. In multicore and manycore systems, multiple concurrent threads can share resources in the system. Multiple hardware threads running on a core share resources of the core such as instruction and data caches, execution units, and a level 2 cache if the core has its own level 2 cache. Modern multicore processors often have a cache shared by all of the threads running on a CPU. Memory bandwidth and bandwidth between CPU sockets are also shared by multiple threads. For a CPU-GPU hybrid system, the PCI Express bus connecting the CPU board to the GPU board is also shared by multiple threads. A GPU core, such as NVIDIA's streaming multiprocessor (SM), spawns a large number of threads, and the threads share registers, shared memory, level 1 cache, and execution units. This high degree of sharing between threads significantly complicates algorithm engineering for multicore and manycore systems.

DRAM and inter-CPU communication bandwidth often become a performance bottleneck in multicore and manycore systems. Algorithm engineering techniques that improve temporal and spatial locality reduce the number of memory accesses that consume DRAM and communication bandwidth. Such techniques reduce not only memory access latency but also bandwidth consumption. Optimization techniques that improve affinity also reduce inter-CPU communication bandwidth consumption and are effective in multicore and manycore systems.

Many applications run a mix of compute-intensive and bandwidth-intensive kernels. In some cases, it is possible to run compute-intensive kernels and bandwidth-intensive kernels at the same time. If we run only compute-intensive kernels, DRAM bandwidth will be significantly underutilized. In contrast, if we run only bandwidth-intensive kernels, DRAM bandwidth becomes a performance bottleneck. Concurrently executing compute-intensive and bandwidth-intensive kernels improves overall performance within a given bandwidth limitation, as demonstrated in [5].

Multiple threads typically share a cache memory in multithreaded architectures. Threads in multicore processors often share an on-chip cache. If there is no data sharing among threads, the threads will compete for the limited cache memory. This significantly reduces the effective cache size per thread. If there is a high degree of data sharing among the threads, the threads can run concurrently with little decrease in the effective cache size. Thus, it is important to run a group of threads with a high degree of data sharing on the same core (with hardware multithreading) or the same CPU to best utilize a shared cache memory.

In multithreaded CPUs, multiple hardware threads share the resources of a core. There are two types of hardware multithreading: temporal multithreading and simultaneous multithreading. In microprocessors with temporal multithreading support, a core can spawn multiple hardware threads but only one thread is active in a given cycle. The hardware threads share a core in a time-sharing fashion. With simultaneous multithreading (or hyperthreading using Intel's terminology), multiple hardware threads are active in a single cycle, and the threads can issue instructions if they have instructions ready and there is an idle execution unit and an available instruction issue port.

Temporal multithreading is effective in hiding the impact of high latency instructions, especially remote memory accesses. Simultaneous multithreading, in addition to the benefit of temporal multithreading, is effective in improving the utilization of a wide-issue core with multiple execution units. For example, a core on the IBM Power7 architecture can issue eight instructions per cycle and has twelve execution units. When only executing independent instructions of a single thread's instruction stream (or in other words, by exploiting only Instruction Level Parallelism (ILP)), we often cannot fully saturate the core's instruction issue ports or execution units. With simultaneous multithreading an execution unit can be utilized if at least one hardware thread on the core has an instruction ready to be executed on the unit. This can be especially effective for vector units. A vector instruction applies an operation to multiple scalar values. Modern microprocessors need high vector unit utilization to achieve high performance, but it can be very difficult to optimize a single thread's instruction stream to fully utilize vector units. With simultaneous multithreading, a vector unit can be exploited if just a single thread on the core has a vector instruction to be executed.

Hardware multithreading is effective in many cases, but if there is a low degree of data sharing among threads, the memory footprint of the application will increase. If a thread does not execute many high latency instructions or is able to saturate a core by only exploiting ILP, hardware multithreading will only increase the contention for cache without a significant performance benefit. The effect becomes more severe for bandwidth-bound applications as the increased cache contention further increases the bandwidth requirement. Given the possibility for adverse effects, hardware multithreading needs to be used only when it is necessary. If long-latency instructions and low core utilization become a performance bottleneck, and the degree of cache contention is low, hardware multithreading is effective. In the opposite case, it may often be better to avoid hardware multithreading.

4 Heterogeneity in Microarchitectures

The increase in the heterogeneity among the microarchitectures of current multicore and manycore processors necessitates the use of algorithm engineering. In some cases, programmers are allowed to build their own system for their target applications; in this case, they need to first select an appropriate combination of architectures for their applications. Even given a heterogeneous system, the work must be partitioned to best exploit the distinct capabilities of the individual computing units in the system. The application should be tuned differently for different microarchitectures as well. To address these issues, programmers need to understand the differences among architectures and their impact on the performance of their applications. This section discusses such issues and algorithm engineering methodologies focusing on typical multicore CPUs and GPUs.

CPUs (such as the Intel Nehalem) and GPUs (the NVIDIA Fermi) support vector instructions, but their vector units have different structures. The Intel Nehalem's vector unit is a single execution unit while the Fermi's vector unit is a group of 16 scalar execution units (NVIDIA refers this scalar unit as a CUDA core). Programmers often need to use compiler intrinsics to exploit the Nehalem's vector unit. To use the Fermi's vector unit to apply operations to 1D, 2D, or 3D arrays, the CUDA programming model allows programmers to map a thread to each element in the array. Then, programmers only need to write code that describes a sequence of operations for a thread without using hard-to-memorize compiler intrinsics. The Nehalem's vector unit width is 16 bytes while the Fermi's vector unit width is 64 bytes[1]. Achieving high utilization of vector units is important and often becomes a major performance tuning issue [7; 10; 13].

Both microarchitectures work well in executing parts of the code that can be fully vectorized. However, if the code cannot be fully vectorized, they work differently. The Nehalem's vector unit is either 100% utilized (if a vector instruction is issued) or 0% utilized (if there is no vector instruction to be issued) in each cycle. With simultaneous multithreading, a vector unit can be utilized if just one hardware thread on the core has a vector instruction for the unit. The CUDA programming model for the Fermi GPU groups 32 threads to form a thread group (or a warp using NVIDIA's terminology). To achieve 100% vector unit utilization, every thread in a warp needs to be on the same execution path. If not, such as with an if-else statement, a vector unit is only partially utilized. Figure 6 illustrates the case.

It is often easier to program for the Fermi's vector unit than the Nehalem's vector unit, but the Nehalem's vector unit is more flexible in applying one-byte and two-byte granularity operations or supporting vector instructions that use multiple scalar elements to compute the final result. The Nehalem's 16-byte wide vector unit can support 16 one-byte additions with a single instruction. The Fermi's 64-byte wide vector unit, in contrast, cannot support 64 one-byte additions in a single cycle. The Fermi's vector unit is also ineffective in supporting operations that access multiple scalar elements in a single vector.

Intel Nehalem's packed sum of absolute differences (PSADBW) instruction provides an example. Assuming two 16 byte input data, this instruction computes the absolute differences of the bytes in the same position of the two input data and sums the eight low differences and eight high differences. The instruction performs 16 one-byte granularity absolute difference computations and sums two groups of eight scalar elements. The Fermi's vector unit cannot support this operation using a single instruction.

A Nehalem core has separate scalar and vector units. A Fermi core, in contrast, does not have a separate scalar unit. Both the Nehalem core and Fermi core work well for highly vectorizable code. If a Nehalem core needs to execute an instruction stream with a mix of scalar and vector operations, the core can execute the scalar operations using scalar units and the vector operations using vector units in the core. Registers can be used to pass data between the scalar units and the vector units in the core. Passing data using registers has very low latency.

If a Fermi core, on the other hand, needs to execute an instruction stream with both scalar and vector operations, the core needs to execute both the scalar and vector operations using its vector units. A Fermi's vector unit consists of 16 scalar units, but only one of the 16 units can be utilized in executing scalar operations. This can significantly lower the performance as a Fermi core has low single-threaded performance. For CPU-GPU hybrid systems, an alternative approach is to run scalar operations using a CPU and run vector operations using a GPU. However, this requires communication between CPU and GPU across the PCI Express bus. This has much higher latency than passing data using registers as in the

[1] Intel's Knights Family Many Integrated Core chip is expected to have a 64 byte wide vector unit.

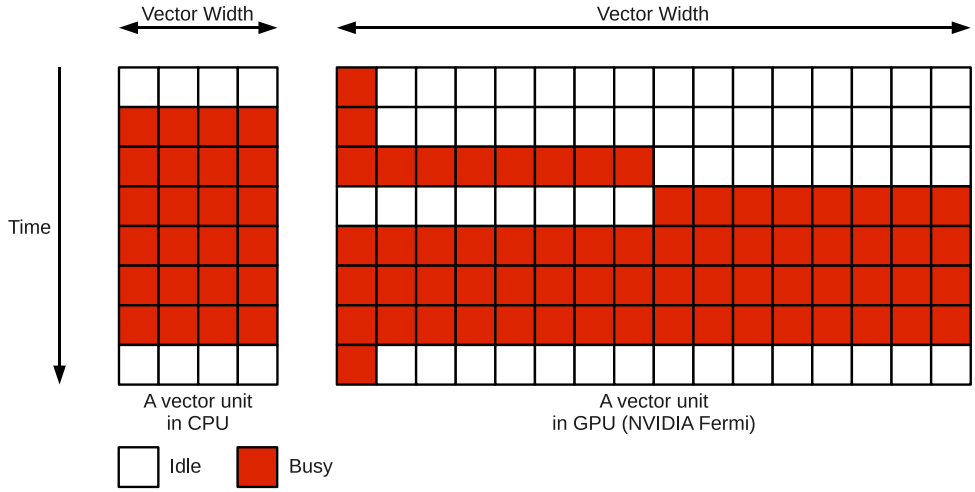


Figure 6 Vector unit utilization in CPU and GPU. A CPU (Nehalem)'s vector unit is either 100% utilized or 0% utilized. A GPU (Fermi)'s vector unit can be partially utilized.

case of the Nehalem. As a result, the Fermi core does not work well in executing instruction streams that frequently switch between scalar operations and vector operations.

Programmers need to consider these differences when selecting architectures for their applications or partitioning work for a heterogeneous system. GPUs are often more effective for highly vectorizable code with mostly four-byte and eight-byte operations. If their target application needs to execute a large number of scalar instructions and many vector instructions for vectors with one-byte or two-byte elements, the Nehalem or the forthcoming Intel Knights family architecture could be a better fit.

The Fermi GPU architecture and typical CPUs have different memory subsystems, and programmers should also consider this difference. Typical CPUs have a cache-based memory subsystem. A GPU based on the Fermi architecture has a hybrid memory subsystem consisting of a cache memory and a software-managed scratchpad memory (or a shared memory using NVIDIA's terminology). Each GPU core has a 64 KB memory and programmers can partition the memory into a 16 KB cache memory and a 48 KB scratchpad memory or a 48 KB cache memory and a 16 KB scratchpad memory. The scratchpad memory is effective for workloads with heavily accessed data shared by threads on a same Fermi core and statically predictable data access patterns. If the heavily accessed data fit into the scratchpad memory, explicitly placing the data in the scratchpad memory is desirable; this prevents heavily accessed data from being evicted from the fast memory by a hardwired cache line replacement mechanism which does not perfectly match with application requirements. Cache memory is a better choice if an application has irregular data access patterns, a memory footprint larger than the scratchpad memory size, and a certain degree of temporal or spatial locality. Programmers need to adjust the partitioning of the hybrid memory and access data using an appropriate data access mechanism to achieve the best performance.

5 Conclusions

Multicore and manycore systems are now ubiquitous in the computing industry. Multicore and manycore processors can enable high performance and power efficient computing but also challenge algorithm engineers as we have discussed in this article.

To achieve both high performance and high productivity, programmers should first understand the match between application requirements and the different architectures that are available. The match between an application's communication patterns and the memory subsystem of the target platform is very important. Recall that frequent CPU-GPU data transfers can easily offset the high computing power of GPUs. Programmers also need to consider the type of necessary operations and how those operations can be supported by target platform architectures, especially their vector units. Architecture specific performance tuning – such as vectorization and the use of architecture specific data transfer mechanisms – often leads to significant performance boost as well but at the cost of additional programming effort.

Algorithm engineering for modern multicore and manycore systems is a multi-objective problem which needs to balance performance, system cost, power, and program development and maintenance cost. This article aims to provide initial guidelines to effectively solve such complex problems.

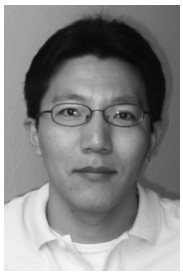
References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, Dec 2006.

[2] D. A. Bader and G. Cong. On the architectural requirements for efficient execution of graph algorithms. In: Proc. of Int'l Conf. on Parallel Processing, pages 547–556, Oslo, Norway, Jun 2005.

[3] D. A. Bader, B. M. E. Moret, and P. Sanders. High-Performance Algorithm Engineering for Parallel Computation. In: Experimental Algorithmics, LNCS 2547, pages 1–23, 2002.
[4] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. In: Proc. of the IEEE, 81(2):211–243, 1993.
[5] A. Chandramowlishwaran, K. Madduri, and R. Vuduc. Diagnosis, Tuning, and Redesign for Multicore Performance: A Case Study of the Fast Multipole Method. In: Proc. of 2010 ACM/IEEE Int'l Conf. on High Performance Computing, Networking, Storage ansd Analysis, New Orleans, LA, USA, Nov 2010.
[6] D. Chaver, M. Prieto, L. Pinuel, and F. Tirado. Parallel wavelet transform for large scale image processing. In: Proc. of the Int'l Parallel and Distributed Processing Symposium (IPDPS), pages 4–9, Ft. Lauderdale, FL, USA, Apr 2002.
[7] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. 2-D Wavelet Transform Enhancement on General-Purpose Microprocessors: Memory Hierarchy and SIMD Parallelism Exploitation. In: Proc. of Int'l Conf. on High Performance Computing, LNCS 2552, pages 9–21, Bangalore, India, Dec 2002.
[8] S. Kang, D. A. Bader, and R. Vuduc. Understanding the design trade-offs among current multicore systems for numerical computations. In: Proc. of Int'l Symp. on Parallel and Distributed Processing, Rome, Italy, May 2009.
[9] A. Kleen. A NUMA API for Linux. Technical Report, Apr 2005.
[10] J. Kurzak, W. Alvaro, and J. Dongarra. Optimizing matrix multiplication for a short-vector SIMD architecture – CELL processor. In: Parallel Computing, 35(3):138–150, 2009.
[11] T. Y. Morad, U. C. Weiser, A. Kolodnyt, M. Valero, and E. Ayguadé. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. In: Computer Architecture Letters, 5(1):14–17, 2006.
[12] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: Proc. of Int'l Conf. on Supercomputing, Reno, NV, Nov 2007.
[13] X.-L. Wu, Y. Zhuo, J. Gai, F. Lam, M. Fu, J. P. Haldar, W.-M. Hwu, Z.-P. Liang, and B. P. Sutton. Advanced MRI reconstruction toolbox with accelerating on GPU. In: Proc. of Conf. on Parallel Processing for Imaging Applications, San Francisco, CA, Jan 2011.

Received: March 3, 2011



Seunghwa Kang completed his Ph.D. at Georgia Institute of Technology in 2011 and joined Pacific Northwest National Laboratory as a postdoctoral researcher. He is also affiliated in Institute for Systems Biology. He has worked on optimizing various kernels and applications for multicore processors and accelerators and analyzing the match between various applications and programming models and architectures. His recent research interest is in solving computational biology problems by designing novel models and algorithms and exploiting high-performance computing.

Address: Georgia Institute of Technology, 266 Ferst Drive, 30332 Atlanta, GA, USA

David Ediger is a Ph.D. student in Electrical and Computer Engineering and a research assistant in the High Performance Computing Lab at Georgia Tech. He completed his B. S. in Computer Engineering in 2008 under the direction of Dr. Tarek El-Ghazawi at The George Washington University focusing on Unified Parallel C (UPC) and reconfigurable systems and an M. S. in Electrical and Computer Engineering from Georgia Tech in 2010. David is principal developer of GraphCT, the graph characterization package for highly parallel, massive graph analysis.

Address: Georgia Institute of Technology, 266 Ferst Drive, 30332 Atlanta, GA, USA

David A. Bader is a Full Professor in the School of Computational Science and Engineering, College of Computing, at Georgia Institute of Technology, and Executive Director for High Performance Computing. Dr. Bader is a lead scientist in the DARPA Ubiquitous High Performance Computing (UHPC) program. He received his Ph.D. in 1996 from The University of Maryland, and his research is supported through highly-competitive research awards, primarily from NSF, NIH, DARPA, and DOE. Dr. Bader serves on the Research Advisory Council for Internet2 and the steering committees of multiple international conferences. Dr. Bader's interests are at the intersection of high-performance computing and real-world applications, including computational biology and genomics and massive-scale data analytics. He is also a leading expert on multicore, manycore, and multithreaded computing for data-intensive applications such as those in massive-scale graph analytics. He has co-authored over 100 articles in peer-reviewed journals and conferences, and his main areas of research are in parallel algorithms, combinatorial optimization, massive-scale social networks, and computational biology and genomics. Prof. Bader is an IEEE Fellow, a National Science Foundation CAREER Award recipient, and has received numerous industrial awards from IBM, NVIDIA, Intel, Sun Microsystems, and Microsoft Research. He served as a member of the IBM PERCS team for the DARPA High Productivity Computing Systems program, was a distinguished speaker in the IEEE Computer Society Distinguished Visitors Program, and has also served as Director of the Sony-Toshiba-IBM Center of Competence for the Cell Broadband Engine Processor.

Address: Georgia Institute of Technology, 266 Ferst Drive, 30332 Atlanta, GA, USA