J. Parallel Distrib. Comput. 74 (2014) 2574–2581

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

What is ahead for parallel computing

Wen-mei Hwu

University of Illinois at Urbana-Champaign, United States

HIGHLIGHTS

- A clear articulation of what makes parallel algorithms hard in multicores and manycores.
- A practical classification of levels of challenges in parallel algorithms.
- An overview of practical techniques for improving parallel algorithm scalability and efficiency.
- A detailed real example of a highly optimized parallel GPU algorithm.

ARTICLE INFO

Article history:
Received 2 October 2013
Received in revised form
13 February 2014
Accepted 13 February 2014
Available online 12 March 2014

Keywords:
Parallel algorithms
Parallel data structures
Locality
Memoryicore bandwidth
GPU
Multicore
Algorithm optimization
Algorithm library
Data layout

ABSTRACT

With the industry-wide switch to multicore and manycore architectures, parallel computing has become the only venue in sight for continued growth in application performance. In order for the performance of an application to grow with future generations of hardware, a significant portion of its computation must be done with scalable parallel algorithms. It is therefore important to develop and deploy as many scalable parallel algorithms as possible. This paper takes a critical look at the major challenges involved in the development of scalable parallel algorithms and points to needs for compiler tool innovations to help address these challenges.

0. Introduction

With the industry-wide switch to multicore and manycore architectures, parallel computing has become the only venue in sight for continued growth in application performance. In order for the performance of an application to grow with future generations of hardware, a significant portion of its computation must be done with scalable parallel algorithms. It is therefore important to develop and deploy as many scalable parallel algorithms as possible.

So, why this is different from what the high-performance computing community has been working on in the past several decades? A simple answer is that the bulk of the prior work was along the lines of performing domain partitioning and use serial algorithms on each partition. While this approach has been very effective in allowing the science community to address increasingly larger problems using increasingly larger clusters, it has not made significant progress in parallelizing the algorithms used in each computing node. A good example is that many Intel Math Kernel Libraries are still only in sequential form when asked to solve a single large problem. What we need to do this time around is to introduce parallel algorithms into each computing node.

The next question is what makes the parallel algorithms for multicore and manycore processors challenging? As it turns out, the main challenges arise from the fact that the rate at which memory data can be accessed by the processor chips is much lower than the rate at which arithmetic operations can be performed on the chip. A closer look into the standard organization of the memory system further reveals that data accesses need to be highly regular in order for the data to be delivered to the processor at a rate close to the advertised rate. Today, these deficiencies are compelling parallel algorithm designers to resort to highly sophisticated data locality and regularization techniques, thereby drastically complicating their algorithms. A legitimate question is whether such deficiency will simply disappear as the technologies advance. Unfortunately, the trend is in the opposite direction, as I will explain in more detail below.

E-mail address: w-hwu@illinois.edu.

http://dx.doi.org/10.1016/j.jpdc.2014.02.005
0743-7315/

1. Three important trends in computing platforms

In general, I see three important trends in technologies and computing platforms. First, the on-chip execution resource is growing faster than the off-chip memory bandwidth. While the on-chip execution resources have been growing with Moore's law, the off-chip memory bandwidth has been held back by the slow evolution of DRAM architecture and the monitory and energy cost of I/O pins of chip packaging. We are at the point where about eight arithmetic operations need to be performed for each byte of data being brought on chip in order to fully utilize the execution capacity of manycore chips. This is in sharp contrast to the long-held rule of one floating-point operation for each byte of data in the High-Performance Computing (HPC) community. Unfortunately, the gap is expected to increase rather than decrease in future generations of computing devices. The reason is that the cost of increasing off-chip memory bandwidth is increasing much faster than the cost of increasing on-chip execution resource. This compels algorithm designers to keep as much data in the on-chip memory as possible for re-use, both across parallel threads and within each thread. Unfortunately, this is an increasingly challenging endeavor due to the next important trend.

The second trend is that the amount of on-chip memory is growing slower than the amount of on-chip execution resource. Unfortunately, traditional cache memories have shown to have diminishing return, where doubling the cache size does not cut the miss rate by two but rather by square root of two [4]. Due to the pressure to provide more computing power to demanding applications, vendors of both CPUs and GPUs are increasing on-chip execution rates faster than on-chip memory. This has incurred high pressure on algorithm designs. For many-core chips, the recent NVIDIA Kepler GK110 GPU has 256 kB of registers, 64 kB of configurable L1/Shared Memory, and 48 kB in each Streaming Multiprocessor (SMX). It also has 1536 kB of L2 cache that is shared among 15 Streaming Multiprocessors [9]. Among all 15 Streaming Multiprocessors, there is a total of 1904 kB of on-chip memory. While this looks like a large number, this amount of on-chip memory is shared by up to 30,720 active threads. This leave us with only 62 bytes of on-chip memory capacity per thread. In many applications, the most effective way of cutting down the use of memory bandwidth is to load tiles of data into the on-chip memory and have all parallel threads to perform their computation on these data from the shared memory. This approach becomes more difficult when the amount of available on-chip memory is small.

The third trend is that vectors are playing an increasingly important role in both memory accesses and arithmetic operations. In particular, vectorized memory accesses are becoming a key to high data delivery rate to manycore processors. This is a direct consequence of the increasing DRAM burst size required to achieve specified data delivery rates. Localized, vector-style data accesses are a well understood programming style that takes advantage of DRAM bursts. Programs that are well vectorized in memory accesses can perform an order of magnitude better than those that are not in today's manycore processors. The difference is expected to increase in the future, which will increasingly compel algorithm designers to regularize the memory access patterns of their algorithms.

2. Life is not fair in parallel computing

Fig. 1 illustrates the efforts to cover parallel portions of the applications with multicore CPU architectures and manycore GPU architectures. The center (core) of the cartoon peach picture represents the sequential portions of the applications. These sequential portions have been the target of modern instruction-level parallelism techniques that wring limited amount of parallelism out of these portions. The latency reduction mechanisms in modern CPUs, including cache memories, branch predictors, and data forwarding are important in preserving the instruction-level parallelism in these portions.

The orange flesh of the cartoon peach represents the data parallel portions of the applications. These portions typically process large data sets whose elements can be processed in parallel. Modern media and data analysis applications tend to have large data parallel portions. The traditional CPUs do not have sufficient amount of execution resources to harvest the data parallelism and achieve dramatic increase in performance. The bump coming out of the peach core depicts the SIMD or vector extensions to the CPU instruction set in order to exploit a much higher level of data parallelism in CPUs. The trend is to increase the width of the SIMD instructions to exploit a higher level of parallelism.

The meshed layer in the peach flesh represents the types of data parallel applications that can be efficiently covered by manycore architectures today. These applications typically are based on parallel algorithms that have high-level of data re-use and vector memory accesses, such as matrix–matrix multiplication, convolution, and particle–grid calculations. The arrows coming out of the layer represent efforts to develop new algorithms and/or to add hardware resources such as more on-chip memory to broaden the types of applications that can be effectively covered by manycore architectures.

The main point of Fig. 1 is that there is a large population of parallel applications that are currently covered neither by CPUs nor manycore processors. Although these applications are rich in data parallelism, they lack the regular accesses and data re-use needed for effective execution by the manycores. The dividing line between haves and have-nots is whether the algorithms used have data re-use and regular accesses. New efforts to design algorithms that exhibit more regular accesses and data re-use are needed for these applications to run effectively on manycore processors. In fact, I argue that the same efforts are also needed for these applications to run effectively on multicore CPUs with wide SIMD extensions.

3. Five levels of challenges in developing parallel algorithms

As we seek to design new parallel algorithms for multicore and manycore processors, we must understand that applications present at least five levels of difficulties. I argue that all these difficulties are equally present whether one programs a manycore GPU or a multi-core CPU. As long as one is trying to achieve high performance, energy efficient execution of highly parallel applications, these challenges must be met. Unfortunately, there is currently little technology in main stream compilers to help programmers to meet these challenges.

The most difficult level is that some applications do not have work-efficient algorithms that have sufficient parallelism. That is, we simply do not know how to solve the problem with a large number of parallel execution units without significantly increasing the computation complexity. Examples of such problems include finding shortest paths between two points in a graph and Delaunay triangulation of a mesh. In the shortest path problem, all known parallel algorithms with high levels of parallelism have significantly higher complexity, or equivalently much lower work efficiency, compared to the most efficient sequential algorithms. For large data sets, the increased computational complexity unfortunately results in so much work that the execution time of parallel algorithms can be slower than sequential algorithms. Unfortunately, large data sets are what make parallel algorithms compelling and thus make the work-inefficient parallel algorithms impractical.

In Delaunay triangulation, we have speculative algorithms that exhibit a small amount of parallelism due to the lack of known methods to divide the mesh into independent parts for large-scale

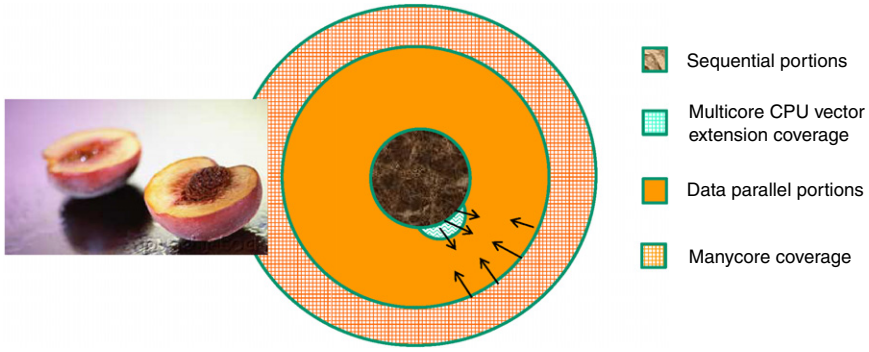


Fig. 1. Coverage of sequential and parallel application portions.

parallel processing [10]. There is simply not enough parallelism to achieve good performance scalability. In many cases, finding work-efficient parallel algorithms or functionally correct problem partition methods has been the target of decades of investigations and would be a fundamental contribution to computer science. While we cannot expect to make broad, rapid progress in addressing this level of challenge, we must continue to support long-term, excellent work in this area.

The second level of challenge is presented by applications that have known parallel algorithms with ample parallelism but questionable numerical stability. Some of these parallel algorithms do not have the same level of numerical stability as well-known sequential algorithms. For example, the Parallel Cyclic Reduction (PCR) algorithm for solving tri-diagonal systems performs independent variable elimination operations on adjacent equations to recursively divide a linear system into independent sub-systems. While PCR by itself is not work efficient, with computational complexity of $O(N \log(N))$, it can form a work-efficient hybrid with the sequential Thomas algorithm. Such hybrid has been shown to exhibit excellent scalability of parallelism with data size [3,5]. However, for some numerical applications, PCR based algorithms are known to be numerically unstable. Such inferior numerical stability has hampered the production use of PCR hybrids. This problem is recently solved by Chang et al. [2], which builds on years of previous work by many researchers on this problem. Their solver is the first one that matches the scalability and speed of the CUSPARSE solver and the numerical stability of the Intel Math Kernel Library (MKL) solver. Similar problems exist in other linear algebra solvers that achieve better problem division by avoiding pivoting. Further work is needed to improve the numerical stability of these parallel algorithms.

The third level of challenge is found in applications that have parallel algorithms but are plagued by catastrophic load imbalance due to highly non-uniform data distribution. For example, there are highly parallel breadth-first search algorithms for graph problems [7]. These algorithms scale well for uniformly distributed graphs where the number of neighbors for nodes does not vary dramatically. However, their performance drops rapidly when the distribution of node connectivity becomes highly skewed [7]. For scale-free graphs, a small number of nodes are connected to a very large number of neighbors. Such non-uniform connectivity is known to cause severe load imbalance and often make parallel algorithms run slower than sequential algorithms. Unfortunately, many real-world graph problems are based on graphs with highly skewed connectivity. Dealing with such load imbalance is difficult. There are limited data pre-processing techniques that can help alleviate the problems with moderate level of load imbalance. However, effective solutions are still yet to be developed for many common, severe cases.

The fourth level of challenge arises in applications where parallel algorithms do not have sufficient data reuse to achieve good scalability. A good example is the sparse matrix–vector multiplication kernel of conjugate gradient solvers. While there is ample parallelism in the algorithm, there is no data reuse in accessing the sparse matrix. While there should be some reuse of the vector elements in theory, the working set is typically too large to be captured by the available on-chip storage. As a result, the achieved performance is typically limited by the available off-chip memory bandwidth, which will not be increasing as fast as the on-chip execution resources. There is active research in communication avoidance algorithms to improve data reuse in these applications. In the context of a conjugate gradient solver, a potential solution is to perform multiple steps of matrix–vector multiplication without performing a residual-gradient calculation in between. There are existing techniques that can organize such polynomial computation of the matrix and the vector to promote data reuse. However, such approach does not have close guidance from the residual-gradient evaluations and thus may end up converging more slowly. Furthermore, it can also affect the numerical stability of the solver. Active research projects are being conducted to address these difficulties.

The fifth level of challenge, and perhaps the least daunting one of the five, is that for parallel algorithms with high levels of parallelism and significant data reuse, engineering the implementation of parallel algorithms to actually achieve good scalability in many-core and multicore processors is still challenging. Today, a parallel programmer needs to determine layout arrangements of data, allocate memory and temporary storage, arrange pointers, perform index calculation, and orchestrate data movement in order to make use of the on-chip memory resources to support data re-use. The programmer also has to decompose work into tasks, organize threads to perform the tasks, perform thread index calculations to access data in different levels of the memory hierarchy, determine data sharing patterns, and check data bounds. Many parameters of these arrangements need to be determined for each hardware platform. All these calculations are tedious and error prone.

4. SGEMM—a “simple” example

For example a simple parallel kernel for Single-precision General Matrix Multiplication (SGEMM) can be written in about 10 lines of code in CUDA or OpenCL. However, the performance will not be anywhere close to what the execution resources of a many-core processor allow. The main issue is low compute to DRAM access ratio. Many DRAM accesses are done for each floating-point calculation. A good implementation will typically have about 2–3 times the code and will execute more than 10 times faster. In general, one needs to eliminate as many reads from and write to DRAM as possible to minimize the consumption of off-chip memory bandwidth. This is commonly achieved by the tiling or blocking the data structure accesses [6]. Such an example is shown in Fig. 2. The extra code (including Lines 11, 14, 15, 16, 17, 20,) is needed to calculate

```
1  #define TILE_N 16
2  #define TILE_TB_HEIGHT 8
3  #define TILE_M (TILE_N*TILE_TB_HEIGHT)
4  __global__ void mysgemmNT( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k, float alpha, float beta ){
5  {
6    float c[TILE_N];
7    for (int i=0; i < TILE_N; i++) c[i] = 0.0f;
8    int mid = threadIdx.y * blockDim.x + threadIdx.x;
9    int m = blockIdx.x * TILE_M + mid;
10   int n = blockIdx.y * TILE_N + threadIdx.x;
11   __shared__ float b_s[TILE_TB_HEIGHT][TILE_N];
12   for (int i = 0; i < k; i+=TILE_TB_HEIGHT) {
13      float a;
14      b_s[threadIdx.y][threadIdx.x]=B[n + (i+threadIdx.y)*ldb];
15      __syncthreads();
16      for (int j = 0; j < TILE_TB_HEIGHT; j++) {
17         a = A[m + (i+j)*lda];
18         for (int kk = 0; kk < TILE_N; kk++) c[kk] += a * b_s[j][kk];
19      }
20      __syncthreads();
21   }
22   int t = ldc*blockIdx.y * TILE_N + m;
23   for (int i = 0; i < TILE_N; i++)   C[t+i*ldc] = C[t+i*ldc] * beta + alpha * c[i];
24 }
```

Fig. 2. A high-performance CUDA SGEMM kernel.

index for tiled data management, synchronize the timing of collaborating threads, and handling the corner cases when the matrix size is not a multiple of the tile sizes. The code is a simplified version of a widely used GPU SGEMM by Volkov and Demmel [1].

Overview of locality management strategy

In order to understand the code in Fig. 2, we need to first understand its locality management strategy shown in Fig. 3. Assume that the function is to calculate $C = A * B$, where A, B, and C are matrices and all C elements will be calculated in parallel. Recall that the element of the output C at row r and column c is the inner product between row r of input A and column c of input B. The locality of the kernel in Fig. 2 is managed by partitioning C into tiles and coordinating the production of C elements within each tile to re-use the input A and B elements when they are fetched from DRAM. Fig. 3 shows a simple, small example where C is partitioned into 4×2 (4 rows and 2 columns) tiles. In this kernel, all C elements in the same horizontal strip of the tile will be processed by one thread. This enables vectorization as well as register tiling within each thread.[1]

Locality management for C

In Fig. 2, Line 1 defines the number of C elements in each horizontal strip as a compile-time constant, 16 in this case. Line 6 declares a local array c[] (lower-case) to hold the C elements to be processed by a thread. Note that since the dimension of the c[] array, TILE_N, is a compile-time constant, all c[] elements can and will be allocated into registers. That is, each thread will accumulate the inner product values into these registers until they are complete. These registers are initialized in Line 7. They will then be used to update the output array C[] (upper-case) elements in DRAM in Line 23. By maintaining the temporary c[] elements in registers, the kernel eliminates the vast majority of the reads from and writes to the C[] elements in DRAM.

The index calculation in Line 22 is quite complex. One must first understand that in CUDA, threads that execute a kernel are

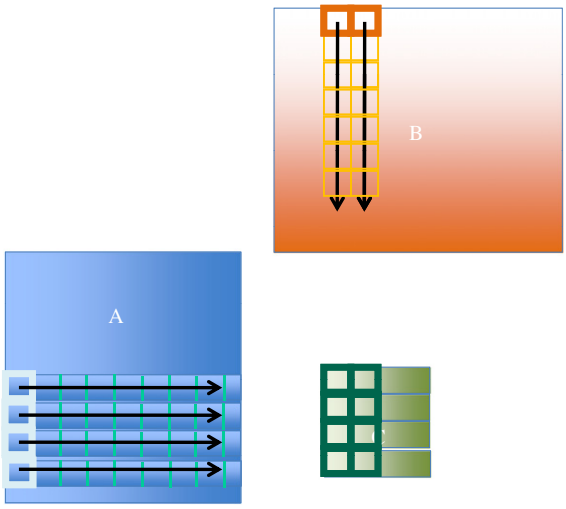


Fig. 3. Register and shared-scratchpad titling for SGEMM.

organized into a two-level hierarchy. At the lower level, multiple threads form a block. All threads in a block can efficiently synchronize with each other through barrier synchronization and share data with each other through on-chip shared scratchpad memories. Each block can be a 1D, 2D, or 3D array of threads. In CUDA, the organization of blocks, or the number of threads in each dimension of the block, is specified at the time the kernel function is called.

The kernel in Fig. 2 assigns one block to process each C[] array tile. The 2D thread block organization is well matched with the 2D matrix data. In a 2D block, each thread can read built-in variables blockDim.x and blockDim.y to get the number of threads in the x and y dimensions of each block. Furthermore, each thread can read built-in variables threadIdx.x and threadIdx.y to get its own index in the x and y dimensions with the block. Lines 8–10 use these built-in variables. Line 8 combines the threadIdx.x and threadIdx.y into a single linear index. For example, if a block is 4×2 (4 in the y dimension and 2 in the x dimension), the value of blockDim.x will be 2. The value of threadIdx.$y * $ blockDim.x + threadIdx.x will be equivalent to threadIdx.$y * 2$ + threadIdx.x, where threadIdx.y varies from 0 to 3 and threadIdx.y from 0 to 1. This assigns a unique

[1] For the discussions of this paper, we use code examples from CUDA C, which is an extension to the C language. The data layout is therefore row-major, meaning that all elements in a row of a matrix will be placed into consecutive memory locations. This is in contrast to the FORTRAN column major layout where all elements in a column of a matrix will be placed into consecutive locations.

linear index to all the 4×2 threads in a block, which ranges from 0 to 7. For example, if threadIdx.$y = 1$ and threadidx.$x = 0$, the linear index mid is 2. For another example, if threadIdx.$y = 2$ and threadIdx.$x = 1$, the linear index mid is 5. The mid value will be used to direct each thread to its assigned matrix elements, as we will discuss in detail below.

Locality management for A and B

The locality management strategy used in the Fig. 2 kernel is illustrated in Fig. 3. The idea is to load a vertical strip of *A* and a horizontal strip of *B* and use them to perform one step of matrix multiplication step for all the *C* elements in a corresponding rectangle. In Fig. 3, two elements of *A* and 4 elements of *B* are loaded and used to calculate multiply-accumulate step of all the *C* elements in the 4×2 rectangle. Without this reuse, one *A* and one *B* element would be loaded for calculating each *C* element. Therefore, the number of memory loads goes down from $2 * 4 * 2 = 16$ to $2 + 4 = 6$. Note that this is a small example. In practice, the sizes of the strips are typically 64 or more for *A* and 16 or more for *B*. For a 64×16 rectangular tile, the number of memory loads goes down from $2 * 64 * 16 = 2048$ to $64 + 16 = 80$, a much more drastic reduction in memory bandwidth consumption.

Now we are ready to analyze more source lines of Fig. 2. The locality management strategy is to reuse each *A* element fetched from DRAM in the calculation of all *C* elements in a horizontal strip of the tile and to reuse each *B* element in the calculation of all *C* elements in a vertical strip. In Fig. 3, the *A* element at the upper left corner of the tile will be used in the calculation of both *C* elements in the top strip of the tile. Since all *C* elements in a horizontal strip will be calculated by the same thread, the thread can load the *A* element into its register and reuse it from that register. This is sometimes referred to as register tiling. In Fig. 3, each *A* element will be used 2 times. Thus, for the four *A* elements, four memory loads are used eight times. Eliminating four out of eight total memory loads for *A* elements. In Fig. 2, local variable a declared in Line 13 is used to hold the *A* element value.

There is another important transformation that must be taken into account in order to understand the code that loads *A* elements. In Fig. 3, each thread will be traversing a row of *A*. The C language adopts row-major layout where elements of a 2D matrix are placed into linear ordering on a row-by-row basis. Multiple adjacent threads will be traversing neighboring rows in parallel. However, this is not a favorable data traversal pattern for current manycore processors; adjacent rows access different DRAM bursts and under-utilize the DRAM bursts and consume more off-chip memory bandwidth [6]. A better traversal pattern is for multiple threads to traverse down neighboring columns. This can be achieved by transposing *A*. After transposition, each thread will traverse down a column, as shown in Fig. 4. The source code in Fig. 2 assumes this matrix *A* has already been transposed before the kernel is called.

The locality management strategy for *B* is for all threads calculating a vertical strip of *C* elements to re-use each *B* element. This requires that *B* elements be placed into the shared on-chip scratchpad memory (SM in CUDA and LDS in OpenCL) whose contents can be accessed by all threads in the same block. The space in the scratchpad memory for holding the *B* elements is allocated in Line 11. It is however, a little surprising that *b_s*[][] array is a 2D rather than 1D array. This has to do with load balancing when loading data. Note that a vertical strip of *A* is typically longer than a horizontal strip of *B* (whose length is limited by register file capacity), and we have as many threads in a block as the number of elements in a strip of *A*. For example, in the small example in Fig. 3, we have 4 elements in a vertical strip of *A*, 2 elements in a horizontal strip of *B*, and 4 threads in a block. The smaller number of *B* elements can cause load imbalance when we load *A* and *B* elements into registers and shared scratchpad.

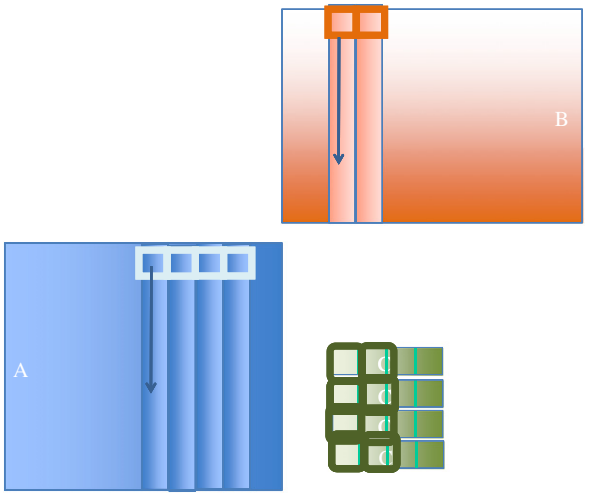


Fig. 4. Matrix *A* is transposed for better DRAM burst utilization.

To mitigate this load imbalance problem, the algorithm loads multiple horizontal strips of *B* rather than a single strip at a time. The number of strips is set to make sure that the same number of *A* elements and *B* elements are loaded in each step of the computation. The ratio between the size of a vertical *A* strip and that of a horizontal *B* is defined as a named constant TILE_TB_HEIGHT in Line 2 of Fig. 2. The value of the constant is set as 8. Since each horizontal strip of *B* has 16 (TILE_*N*) elements and each vertical strip of *A* has $16 * \text{TILE\_TB\_HEIGHT} = 16 * 8 = 128$ elements. This size of the vertical strip of *A* is defined as a name constant TILE_*M* in Line 3. So, in Fig. 3, TILE_*M* is the number of elements in a vertical strip of *A* and TILE_*N* is the number of elements is a vertical strip of *B*.

Line 12 in Fig. 2 contains the header of the main loop for each thread. This loop will iterate through all the steps needed for calculating the final result for each *C* element. Keep in mind that there are TILE_*M* threads in the same block. In each iteration of the loop, a thread loads TILE_TB_HEIGHT elements of *A*, loads one element of *B*, and performs TILE_TB_HEIGHT steps for each of the TILE_*N* *C* elements assigned to it. In terms of the small example in Fig. 4, there are four threads in a block. Each thread will load two vertically neighboring elements of *A* (transposed version), load one element of *B* from two adjacent horizontal strips, and perform two multiply-accumulate steps for each of the two *C* elements assigned to it.

In preparation for all the index calculations for the memory loads, Line 9 defines variable *m* that contains the column position of the first element of *A* to be loaded by the thread. With each block covering one horizontal strip, the *m* value for a block should start with blockIdx.$y * \text{TILE\_}M$. Each thread can then add its mid to form the final *m* value. This gives the statement $m = \text{blockIdx.}y * \text{TILE\_}M + \text{mid}$ in Line 9.

Line 10 defines variable *n* that contains the column position of the first element of *B* to be loaded by the thread. As shown in Fig. 4, in each row of *B*, each block covers one horizontal strip of size TILE_*N*. Therefore, the *n* value for a block should start with blockIdx.$x * \text{TILE\_}N$. Each thread can then add its threadIdx.*x* value to for the final *n* value. Note that a thread block loads TILE_TB_HEIGHT horizontal strips of *B*. We will come back to this point later.

In each iteration of the for loop of Line 12, Line 14 loads TILE_TB_HEIGHT of horizontal strips into the shared scratchpad memory. Since the thread block is organized as a 2D TILE_TB_HEIGHT $\times$ TILE_*N* array of threads, we can use each row to load the TILE_*N* elements in a horizontal strip. Since each strip is a full row away from the previous one, the index calculation takes the form of $B[n + \text{threadIdx.}y * \text{ldb}]$, where ldb is the lowest (*x*) dimension of *B*. This ensures that threads from different rows will load different

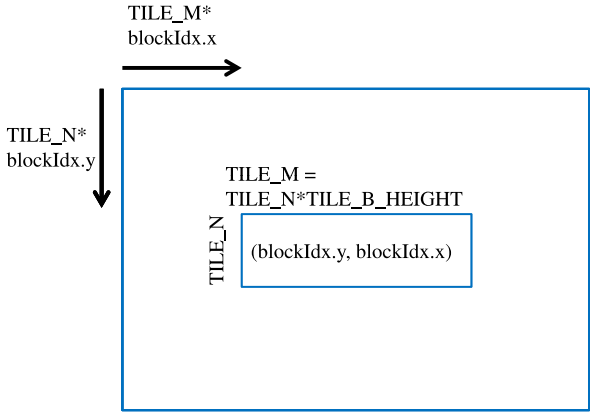


Fig. 5. Tiling details for output matrix C.

horizontal strips of *B*. Since *n* already accounts for the threadIdx.*x* value, each thread in a given row will be loading a different element in the same horizontal strip. Therefore, every *B* element in the set of horizontal strips is loaded into the shared scratchpad by a thread in the block. The value of the induction variable *i* is incorporated into the index of $B[n + (i + \text{threadIdx}.y) * \text{ldb}]$ to allow the set of strips to move down as the loop iterates.

Line 15 is a barrier synchronization to ensure that all threads in the block have finished loading their respective *B* elements into the scratchpad. This is because the *B* elements are used by multiple threads. One must ensure that the *B* elements are in place before any of the threads start to use them from the shared scratchpad. Also, the barrier synchronization in Line 20 ensures that all threads have finished using the *B* elements in the scratchpad memory before any of them move ahead and overwrite them with the next batch of *B* elements loaded in the next iteration.

Lines 16–19 is a loop that computes TILE_TB_HEIGHT steps of multiply-accumulate for all *C* elements in the tile. Recall that each element is re-used in calculating all TILE_*N C* elements in a horizontal strip in Fig. 4. Each thread thus loads a single *A* element in Line 17. The index for the load is $m + (i + j) * \text{lda}$, where *m* is the column index of the *A* element in the first strip. The value of $(i + j) * \text{lda}$ allows each iteration to move down to the next strip in Fig. 4. The innermost loop uses the *A* element in register (a) to perform one step for all the *C* elements assigned to the thread.

It should be clear that the index calculation for all the loads and stores are quite complex, involving sophisticated mapping from thread indices to array indices. It should also be clear that the complexity is due to the need for managing locality and to reduce memory consumption. From my experience, similar observations can be made in a wide variety of algorithms: the base parallel kernel is fairly simple and the vast majority of the complexity comes from the need to manage locality. This is a fertile ground for good programming models and tools to alleviate the burden on programmers. Tools that strike a good balance between programmer control and convenience will likely make major impact on the productivity of parallel programmers and quality of parallel applications in the future (see Fig. 5).

5. Algorithm techniques for scalability

The tiling optimization we showed in the DGEMM example is only one of the eight commonly used algorithm optimization techniques [11], or optimization patterns, for writing scalable, fine-grain parallel code. Fig. 6 shows these eight optimization techniques and the scalability bottlenecks that they are designed to alleviate. Currently, all these techniques add complexity in a similar way to what we saw in the DGEMM example.

Scatter-to-gather transformation reorganizes the memory updates of parallel threads so that they update distinct sets of outputs. This optimization is typically applied to parallel computation where each thread inspects a subset of inputs and scatters their effect on output points. The optimized computation has each thread to collect all the input effect for a subset of the output points, which is sometimes referred to as *owner computes* in the Message Passing Interface (MPI) programming literature. By assigning each thread to a distinct subset of outputs, the optimization reduces update contention to the outputs. The transformation is usually in the form of loop interchange. While this is conceptually a relatively simple transformation, the implementation can be complex, tedious, and error prone for real applications with large complex loop bodies and early loop exit conditions. Additional complexity usually arises due to the need to perform input binning (see below) operation on inputs to enable work-efficient execution of gather-style algorithms. Robust loop transformation tools and input binning libraries can greatly reduce the programming cost of this type of transformation.

Privatization replicates output data structure to reduce update contention. This is commonly used on computations where output cannot be statically determined, such as histograms. To ensure proper incorporation of contributions from multiple parallel threads, atomic operations are typically used in output updates. Atomic operations, however, tend to have very low throughput and long latency compared to regular memory operations. If multiple atomic operations update the same location, they can effectively serialize execution and reduce scalability. Privatization generates multiple copies of output to reduce the number of collisions during parallel execution. At the end of the computation, these versions are merged back together. Privatization tends to be a straightforward transformation when the output data set is small. Additional complexity arises when one can only afford to privatize a portion of a large output data set. In this case, the programmer needs to identify the most important portion to the optimized code needs to test if the intended output falls within the privatized portion or not.

Thread Coarsening consolidates the work of multiple data-parallel threads into a single thread. This is usually done to enable effective use of vector hardware and processor registers. The transformation is in the form of loop unrolling, loop stripmining, and loop fusion. Fig. 2 is an example of thread coarsening with respect to matrix *A* elements. While this transformation is conceptually simple, it does increase the code size and require tedious adjustments to the use of indices. Furthermore, in order to achieve vectorization and register tiling, one needs to rename variables and deal with potential control divergence of work collected from different threads. Additional complexity arises when the data size is not guaranteed to the multiples of the vector width. With vectors and register files playing increasingly important roles in achieving performance and scalability goals for processors, compiler tools to reduce the effort for thread coarsening will be increasingly valuable.

Data Layout adjusts the arrangement of data in memory to match the need of algorithms and hardware. The fundamental problem is that the DRAM and the caches strongly favor data access with spatial locality. Memory accesses made close to one another in time need to be spatially local to one another in order to achieve a reasonable fraction of the peak bandwidth of DRAMs and caches. Unfortunately, many frequently used algorithms can result in little spatial locality. For example, accesses to logically adjacent elements of the same column of an array in a C program will have little spatial locality because of the row-major layout convention of the C programming language. It is well known that high-performance linear algebra libraries often require that some of the input matrices to be transposed to improve the spatial locality of their algorithms. A related need for data layout adjustment is the use of

Techniques	Memory Bandwidth	Update Contention	Load Balance	Regularity	Efficiency
Scatter to Gather		X			
Privatization		X			
Tiling	X				X
Coarsening	X	X			X
Data Layout	X	X			X
Input Binning	X				X
Regularization			X	X	X
Compaction	X		X	X	X

Fig. 6. Algorithm techniques for locality and scalability.

array of structures. Real-world data are often multi-facet in nature. A video pixel has R, G, and B components. Atoms or electrons in a molecular dynamic system are characterized with their x, y, z locations along with their mass, speed, and charge. External data formats and common libraries store these components in C structures to main logical organization of the data. However, vectorization often prefers separating these components into different arrays.

Another related need for data layout adjustment stems from domain decomposition for massively threaded execution. The decomposed data tend to be in localized chunks of memory. However, the contemporary accesses by the parallel threads are spread across these chunks. A much better arrangement is to interleave the data layout of the chunks so that all contemporary accesses are spatially local to one another. Such change has pervasive effect on the source code and can introduce many errors. Also, efficient conversion or marshaling between these data layout organizations require highly optimized algorithms. Compiler tools and standardized data marshaling libraries can drastically reduce the programming efforts in this area.

Data layout needs are by no means confined in matrix computation. Scalability of graph algorithms can be significantly enhanced with data layout that matches the access patterns of computation [8].

Input Binning, or spatial sorting, is typically performed in conjunction with scatter-to-gather transformation. During a gather-style execution, each thread collects the contribution of all relevant input data to an output point. Unfortunately, in most applications, input data tend to be irregular and have non-uniform, sparse distribution. As a result, the input data tend to explicitly identify their location as well as physical characteristics such as mass or charge. For example, the representation of input atoms in a molecular dynamic system typically carries each atom's position coordinates as well as charge. As a result, one cannot easily determine if an input data element is relevant to an output data point without examining part of its content. However, having all threads to examine all inputs can be a cause for the computational complexity to explode and destroy the work efficiency of parallel algorithms. Input binning is a preprocessing technique that places input elements into well-structured bins in the modeled physical space. Each thread can rule out the bins that are known to be too far away to contain any input element that can be relevant to its output data points. Effective input binning requires sophisticated bin organization to minimize the number of bins examined as well as the number of irrelevant input elements in each bin. High-quality input binning libraries and frameworks can significantly reduce the programming cost of input binning.

Regularization is a technique that improves load balance by offloading the work of the most burdened threads to others. This is often done in conjunction with input binning. As one pre-processes the input data into bins, he/she can set a limit on the number of input elements that can go into each bin. Since the threads will be taking their input from the bins, such limit will mitigate the degree of load imbalance among threads. Regularization can also be in the form of input reshaping or pre-conditioning. For example, for graph algorithms, one can replicate the nodes that have the most number of neighbors, with each replicated node taking a subset of the neighbors. For scale-free graphs, such replication can be crucial for scalable performance of parallel algorithms. Regularization typically requires sophisticated coordination of data structures and should be supported by library functions.

Compaction is a technique that transforms a dense representation of data into a sparse representation to conserve memory bandwidth and storage. This is often performed in conjunction with input binning. Due to the non-uniform nature of input data, there is a need for balancing between the need for easy parallel access to input data and the amount of wasted space in each bin. Techniques that are employed in high-speed sparse matrix representation formats can be generalized to reduce wasted space while maintaining efficient access structure for the input data. This transformation will become even more important due to the growing problem of memory bandwidth and size limitations in exascale computing devices. This is also an area where some of the data layout transformations need to be deployed to preserve vectorizability after compaction. Good libraries and compiler transformation support can significantly improve the software engineering productivity in this area.

To summarize, it should be clear that all the algorithm optimization techniques overviewed in this section are above and beyond parallelism. Having a parallel algorithm for a problem is often time only a part of the parallel programming problem. It should also be clear that the lack of strong compiler transformation and library support for these optimization techniques has been a major source of the high cost of parallel programming in real-world application development.

6. Conclusion

As the computer industry moves ahead, I argue that the most rewarding undertaking is to produce a scalable software code base. Scalable code base is required for applications to experience continued performance and functionality growth in the years to come. In contrast to what many believe, I argue that finding massive

amount of parallelism in many applications is not the real challenge. Optimizations for managing data locality, reducing conflicts and regularizing work distribution are in fact the most critical, challenging, and error prone aspects of producing scalable software. After years of work, we have identified the most commonly used techniques for such optimization. I believe that the most beneficial work is to center the design of new programming languages, libraries, compilers, and optimization tools on these techniques in the context of real-world applications. We can leave a lasting, valuable legacy to the generations to come.

Acknowledgments

At the risk of missing some important ones, I would like to acknowledge the following individuals who have helped to shape my understanding of the problems and solutions overviewed in this article: D. August (Princeton), S. Baghsorkhi (Illinois), N. Bell (NVIDIA), D. Callahan (Microsoft), L. Chang (Illinois), J. Cohen (NVIDIA), T. Conte (Georgia Tech), B. Dally (Stanford), E. Davidson (Michigan), J. Demmel (Berkeley), C. Davis (MulticoreWare), P. Dubey (Intel), M. Frank (Intel), M. Garland (NVIDIA), I. Gelado (UPC), B. Gropp (Illinois), M. Gschwind (IBM), R. Hank (Google), J. Hennessy (Stanford), P. Hanrahan (Stanford), M. Houston (AMD), T. Huang (Illinois), H. Hunter (IBM), D. Kaeli (NEU), K. Keutzer (Berkeley), H. Kim (Illinois), D. Kirk (NVIDIA), D. Kuck (Intel), S. Lumetta (Illinois), S. Mahlke (Michigan), T. Mattson (Intel), N. Navarro (UPC), J. Owens (Davis), D. Padua (Illinois), J. Patel (Illinois), S. Patel (Illinois), Y. Patt (Texas), D. Patterson (Berkeley), K. Pingali (Texas), C. Rodrigues, S. Ryoo (Synopsys), C. Rodrigues (Illinois), K. Schulten (Illinois), B. Smith (Microsoft), M. Snir (Illinois), I. Sung (Illinois), P. Stenstrom (Chalmers), J. Stone (Illinois), S. Stone (Harvard) J. Stratton (Illinois), J. Torrellas (Illinois), M. Valero (UPC), X. Wu (Illinois). I would like to also thank Per Stenstrom for his insightful review comments that helped to significantly improve this article.

References

[1] G. Ballard, J. Demmel, O. Holtz, O. Schwartz, Minimizing communication in numerical linear algebra, EECS Technical Report UCB/EECS-2011–15, February 28, 2011.

[2] L. Chang, J.A. Stratton, H. Kim, W.W. Hwu, A scalable, numerically stable tridiagonal solver using GPUs, in: The International Conference for High-Performance Computing Networking, Storage, and Analysis, SC'12, Salt Lake City, 2012.

[3] A. Davidson, Y. Zhang, J.D. Owens, An auto-tuned method for solving large tridiagonal systems on the GPU, in: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, May 2011.

[4] A. Hartstein, V. Srinivasan, T.R. Puzak, P.G. Emma, On the nature of cache miss behavior: is it $\sqrt{2}$? J. Instr.-Level Parallelism 10 (2008).

[5] H.-S. Kim, S. Wu, L. Chang, W.W. Hwu, A scalable tridiagonal solver for GPUs, in: 40th International Conference on Parallel Processing, ICPP2011, Taipei, Taiwan, September 15, 2011.

[6] D. Kirk, W. Hwu, Programming Massively Parallel Processors—A Hands-on Approach, second ed., Morgan Kaufmann Publisher, ISBN: 0124159923, 2012.

[7] L. Luo, M. Wong, W. Hwu, An effective GPU implementation of breath-first search, in: IEEE Design Automation Conference, DAC, 2010.

[8] D. Merrill, M. Garland, A. Grimshaw, Scalable GPU graph traversal, in: Proc. PPoPP 2012, February 2012.

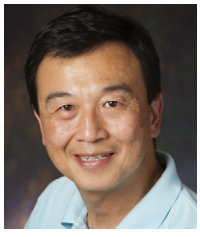
[9] NVIDIA, "NVIDIA's next generation CUDA™ compute architecture: Kepler™ GK110—the fastest, most efficient HPC architecture ever built", http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[10] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M.A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, X. Sui, The tao of parallelism in algorithms, in: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, 2011.

[11] J.A. Stratton, C. Rodrigues, I.R. Sung, L. Chang, N. Anssari, G.D. Liu, W.W. Hwu, N. Obeid, Algorithm and data optimization techniques for scaling to massively threaded systems, IEEE Comput. (2012) 26–32.

Further reading

[1] V. Volkov, J. Demmel, Benchmarking GPUs to tune dense linear algebra, in: The International Conference for High-Performance Computing Networking, Storage, and Analysis, SC'08.



Wen-mei Hwu is a Professor and holds the Walter J. ("Jerry") Sanders III-Advanced Micro Devices Endowed Chair in Electrical and Computer Engineering of the University of Illinois at Urbana-Champaign. His research interests are in the area of architecture, implementation, and compilation for parallel computer systems. He directs the IMPACT research group (www.crhc.uiuc.edu/Impact). For his contributions, he received the ACM SigArch Maurice Wilkes Award, the ACM Grace Murray Hopper Award, and the ISCA Most Influential Paper Award. He is a fellow of IEEE and ACM. Hwu received his Ph.D. degree in Computer Science from the University of California, Berkeley in 1987.