An approach for semiautomatic locality optimizations using OpenMP

Jens Breitbart[1]

[1]Research Group Programming Languages / Methodologies
Universität Kassel
Kassel, Germany
jbreitbart@uni-kassel.de

Abstract. The processing power of multicore CPUs increases at a high rate, whereas memory bandwidth is falling behind. Almost all modern processors use multiple cache levels to overcome the penalty of slow main memory; however cache efficiency is directly bound to data locality. This paper studies a possible way to incorporate data locality exposure into the syntax of the parallel programming system OpenMP. We study data locality optimizations on two applications: matrix multiplication and Gauß-Seidel stencil. We show that only small changes to OpenMP are required to expose data locality so a compiler can transform the code. Our notion of tiled loops allows developers to easily describe data locality even at scenarios with non-trivial data dependencies. Furthermore, we describe two new optimization techniques. One explicitly uses a form of local memory to prevent conflict cache misses, whereas the second one modifies the wavefront parallel programming pattern with dynamically sized blocks to increase the number of parallel tasks. As an additional contribution we explore the benefit of using multiple levels of tiling.

1 Introduction

In the last years multicore CPUs became the standard processing platform for both end user systems and clusters. However, whereas the available processing power in CPUs continues to grow at a rapid rate, the DRAM memory bandwidth is increasing rather slowly. CPUs use multiple levels of fast on-chip cache memory to overcome the penalty of the slow main memory. Caches, however, are only useful if the program exposes data locality, so reoccurring accesses to the data may be fetched from cache instead of main memory.

Parallel programming without considering data locality provides far from optimal performance even on systems based on Intel's Nehalem architecture, which provides a highly improved memory subsystem compared to all previous Intel architectures. In this paper we use two well-known applications – a matrix-matrix multiplication and a Gauß-Seidel stencil – to demonstrate tiling and its effect. We utilize two levels of tiling to increase performance in scenarios for which single-level tiling is not already close to the maximum memory/cache

Test	copy	scale	add	triad
Bandwidth (MB/s)	18463	19345	17949	18356

Table 1. Memory bandwidth of the test system measured with the STREAM benchmark.

bandwidth. Explicit tiling is a technique to overcome conflict cache misses. Explicit tiling requires duplicating data, but still offers an overall increase in performance. Furthermore we provide an example to demonstrate how the shown locality optimizations can be integrated into e. g. the OpenMP syntax with only small changes. The approach however is not limited to OpenMP and could be implemented into other pragma-based systems or as a standalone system, as well. As a result of our OpenMP enhancements, developers must only hint the compiler at what kind of changes are required to achieve the best performance. We expect this to ease locality optimizations. At the time of this writing, the changes have not been implemented in a compiler, but the required transformations are described in detail.

The paper is organized as follows. First, Sect. 2 describes the hardware used for our experiments. Our applications, and the used locality optimizations and results are shown in Sect. 3 for matrix multiplication and in Sect. 4 for the Gauß-Seidel stencil, respectively. Section 5 describes the changes to OpenMP to allow compilers to automatically change code the way we have described in Sects. 3 and 4. The paper finishes with an overview of related work and conclusions, in Sects. 6 and 7, respectively.

2 Hardware setup

All experiments shown were conducted at an Intel Core i7 920, which is a CPU based on the Nehalem architecture. It is a quad core CPU running at 2.67 GHz using a three-ary cache hierarchy. The level 3 cache can store up to 8 MB of data and is shared by all cores of the CPU. In contrast every core has its own level 2 (256 KB) and level 1 cache (32 KB for data). The caches are divided into cache lines of 64 bytes, so in case a requested data element is not in the cache, the system fetches the cache line storing it from main memory. Both level 1 and level 2 cache are 8-way associative. The level 3 cache is 16-way associative. The Nehalem is Intel's first x86 architecture featuring an on-chip memory controller to lower memory access latency. Table 2 shows memory bandwidth measured with the STREAM benchmark [McC95] using 4 threads. We used three memory channels supplied with DDR3-1333 memory modules. Each CPU core provides additional hardware support to run up to two threads efficiently (SMT), so the CPU supports up to 8 threads effectively.

3 Matrix multiplication

We describe our first set of locality optimizations, for which OpenMP pragmas are shown in Sect. 5. Their performance is shown by implementing them in a

trivial matrix-matrix multiplication calculating $A * B = C$ with A, B, and C being square matrices of size n. Our trivial implementation uses three nested loops, which are ordered `i-k-j` with the `i-` and `j`-loop looping over C and the `k`-loop being used to access a column/row of A/B. This loop order allows for easy automatic vectorization and good data locality, for details see e. g. [Wol95]. The parallelization of the matrix multiplication is obvious, as each matrix element can be calculated independently.

The trivial matrix multiplication algorithm is well-known to have low memory locality, since successive accesses to elements of A, B are too far away to keep data in cache [Wol95]. A well-known optimization to improve data locality and cache reuse in such scenarios is tiling [BGS94], also known as loop blocking. Tiling divides the loop iteration space into tiles and transforms the loop to iterate over the tiles. Thereby a loop is split into two loops: an outer loop which loops over tiles and an inner loop which loops over the elements of a tile. We applied loop tiling to our matrix multiplication and refer to the tiles as A_{sub}, B_{sub} and C_{sub}, respectively.

Tiling improves the performance of our matrix multiplication in almost all cases. Measurements (not depicted here) show that there are hardly any L2 cache misses, but the performance suffers from regular L1 cache misses. The phenomenon is measured at any block size, even when choosing a block size that fits into the L1 data cache and its impact is at maximum, when the matrix size is a power of two. Therefore, we expect these misses to be conflict misses, as array sizes of a power of two are worst case scenarios for current CPU caches [CSG98]. Conflict misses arise due to the associativity of the cache, meaning that there are multiple data elements in A_{sub} and B_{sub} that are mapped to the same position in the cache and thereby replace each other even though there is enough space in the cache available. To resolve this problem we propose a new optimization technique called explicit blocking that is inspired by the OpenCL programming system. OpenCL uses local memory that is local to processing units and allocated in on-chip memory. In explicit blocking we allocate an array to store the data used during the calculation – in our example we allocate an array of size $blocksize * blocksize$ to store both A_{sub} and B_{sub} and copy the submatrix from A/B to the newly allocated array. In the calculation we only access the newly allocated arrays. We cannot allocate these blocks in on-chip memory in current CPUs, however by using this technique we no longer suffer from conflict cache misses during the calculations.

To improve the performance further, we added a second level of tiling. We refer to this optimization as multilevel tiling. We combined small explicitly tiled blocks that fit into L1 cache with larger tiles that help to increase data locality in L2 and L3 cache. Our final implementation thereby uses large blocks that are split into smaller subblocks.

Figure 1 shows the performance of our code with the matrix data being stored in a one dimensional array, compiled with the Intel Compiler version 11.1. The performance was measured using matrices of size 8192^2 at both single and double precision. We compare our results with two libraries: Intel's Math Kernel

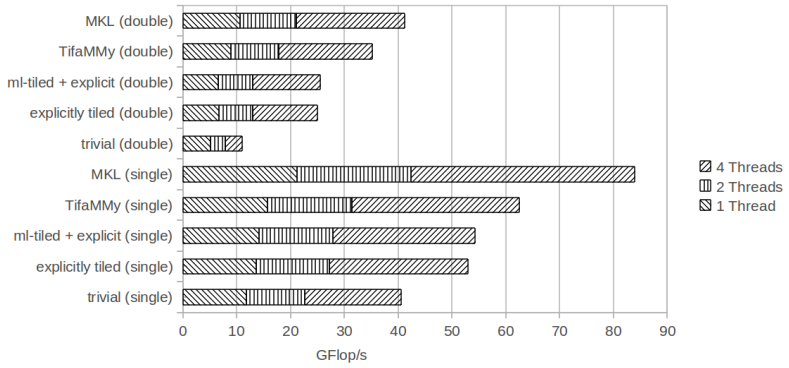


Fig. 1. Matrix multiplication performance (matrix size: 8192^2)

Library[1] (MKL) and TifaMMy [BFGH07]. Both libraries use hand-tuned code and thereby obviously outperform our implementation, which purely relies on the Intel compiler to generate optimized code. TifaMMy has not been optimized for Nehalem-based CPUs and therefore falls behind Intel's MKL. Our trivial version consists only of three nested loops, however the compiler automatically applies tiling to the loop.

The code changes to achieve the performance increase are rather trivial, but yet require writing multiple lines of code. For example applying explicit tiling requires developers to create two new arrays and to copy data. Even though the implementation is simple, it is still a source of bugs. Furthermore, the compiler did not automatically detect that these transformations should be applied and OpenMP does not offer us a way to express these. Some compilers offer ways to specify loop tiling, however using compiler specific options obviously is not-compiler compatible and even interfere with the OpenMP parallelization in an unpredictable way.

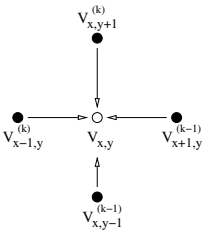
4 Gauß-Seidel stencil

The calculations of the Gauß-Seidel stencil are being applied on a two-dimensional data matrix V with the borders having fixed values. The non-border element with the coordinates (x, y) in the kth iteration is calculated by

$$V_{x,y}^{k} = \frac{V_{x-1,y}^{k} + V_{x+1,y}^{k-1} + V_{x,y-1}^{k-1} + V_{x,y+1}^{k}}{4}.$$

This calculation is repeated for a fixed number of steps or until it converges. Considering the low arithmetic intensity of the calculations, it should come at no surprise that the runtime is limited by memory bandwidth and not by the available processing power. We consider a bandwidth-limited application an interesting scenario, as the gap between available processing power and memory

[1] http://software.intel.com/en-us/intel-mkl/



(a) Gauß-Seidel data dependencies

1	2	3
2	3	4
3	4	5

(b) Wavefront pattern

Fig. 2. Gauß-Seidel

bandwidth is increasing and more scenarios will become bandwidth-limited in the future.

We measure the performance of the Gauß-Seidel stencil by stencils computed per second (Ste/s). We again applied the locality optimizations described in the last section, to our sequential implementation and as a result we increased the performance from 283 to 415 MSte/s.

Figure 2(a) shows a visualization of the data dependencies of the Gauß-Seidel stencil. It is important to notice that the upper and the left values are from the current iteration, whereas the right and bottom value are from the previous iterations. The parallelization requires using the wavefront pattern [Pfi98]. In a wavefront, the data matrix is divided into multiple diagonals as shown in Fig. 2(b). The elements in one diagonal can be calculated in parallel. Tiling can again be applied to increase data locality by creating tiles of V.

We implemented both a strict and relaxed version of the wavefront pattern. The strict version directly follows the wavefront pattern and only calculates the diagonals in parallel, whereas the relaxed version breaks up the diagonals. In the relaxed version we split up the matrix in x-dimension in multiple columns and assign these columns to threads with a round robin scheduling and use one counter per column, which indicates how much of that column has already been updated in the current iteration. These counters are shared by all threads and are used to identify how *deep* the current thread can calculate the current column, before it has to wait on the thread calculating the column left from it. We have implemented two different ways to prevent the race conditions at accessing the shared counters. The first version uses OpenMP lock variables to guard the counters. See Listing 1 for the source code. The second version uses an atomic function provided by the host system to update the counters. We cannot use OpenMP atomic operations as OpenMP does not allow threads to read a variable that has been updated by another thread without synchronizing. When using the host system atomic operation to update the counter, the read must only be joined by a flush/fence. Tiling to x- and y-dimensions is applied in both versions. Tiling to the y-dimension reduces the number of times the shared counters are updated, so larger tiles decrease the number of lock operations,

Algorithm 1 Manual blocked Gauß-Seidel

```
1  int nb_blocks = size/block_size;
2  int *counter = new int[nb_blocks+1];
3  counter[0] = size-1;
4  //initialize all other counters with 0
5  omp_lock_t *locks = new omp_lock_t[nb_blocks+1];
6  //call omp_init_lock for all locks
7  #pragma omp parallel for
8  for (int x=1; x<size-1; x+=block_size) {
9    int y = 1;
10   const int x_block = x/block_size;
11   while (y<size-1) {
12     omp_set_lock (&locks[x_block]);
13     const int lcounter = counter[x_block];
14     omp_unset_lock (&locks[x_block]);
15     for (; y<lcounter; y+=block_size) {
16       for (int xx=x; xx<x+block_size; ++xx)
17         for (int yy=y; yy<y+block_size; ++yy)
18           V[yy][xx] = (V[yy][xx-1] + V[yy][xx+1] +
                 V[yy-1][xx] + V[yy+1][xx])/4;
19       omp_set_lock (&locks[x_block+1]);
20       counter[x_block+1] += block_size;
21       omp_unset_lock (&locks[x_block+1]);
22     }
23   }
24 }
```

however also reduces the chance of having the rows y, $y-1$ and $y+1$ in the cache.

To achieve better performance, we applied multi-level tiling. We continue to use large tiles to reduce the number of lock operations and subdivide the large tiles into smaller tiles to expose the best data locality. However multilevel tiling increases the overhead and decreases performance when running with 8 threads.

As a final optimization we moved from fixed to dynamic tile sizes for the large tiles. Our dynamic implementation chooses smaller tiles at the beginning and the end of the data matrix and larger ones in the middle. Dynamic tile sizes allow more work to be done in parallel, since e. g. the calculation of the second column can only be started if the first tile of the first column is calculated. However, this optimization imposes additional overhead, while on our test system the performance loss due to data dependencies is rather small. As a result, the performance is almost identical to the one using static tile sizes. We expect dynamic tile sizes to be useful on systems with more cores and additional memory bandwidth to satisfy all cores.

Figure 3 shows the achieved bandwidth with the different versions for calculating a matrix of size 16386^2 with 100 iterations. We use double precision for our calculations. Our fully optimized implementation sustains a performance of over 1000 MSte/s. In the best case, the Gauß-Seidel stencil requires both

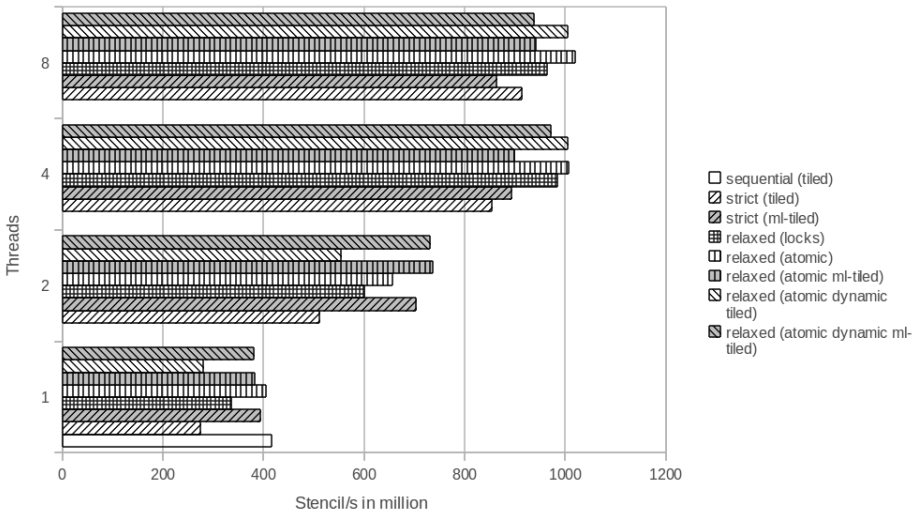


Fig. 3. Gauß-Seidel performance

one element read and one written to/from main memory per stencil. With 16 bytes transfered between the CPU and main memory our implementation uses a bandwidth of over 16 GB/s, which is about 83% of the performance measured with the STREAM benchmark (Tab. 2). The best performance is achieved when using 8 threads with single level tiling. The additional overhead of multilevel tiling reduces performance with 4 or 8 threads, however the increased data locality helps to outperform single level tiling with less threads. We can see that using locks instead of atomic operations decrease our performance by about 6% when using 4 cores. With one thread the parallel atomic version performs almost identical to the sequential version.

5 Locality-aware OpenMP syntax

In the last two sections, we have described our experiences with locality optimizations in two scenarios. The first scenario was matrix-matrix multiplication, which only consists of a perfect loop nest and a trivial parallelization. The loop transformations could have automatically been applied by a compiler, however our compilers did not. Our second scenario also benefits from tiling by both increased data locality and decreased number of lock operations. The parallelization of the second scenario was more complex and the code is not a perfect loop nest. We expect that it is hardly possible for a compiler to identify all required optimizations, however we show in the rest of this section that the OpenMP syntax can be enhanced to let developers tell the compiler which optimizations should automatically be applied. Furthermore we sketch how compilers can do the required transformations and discuss the impact on other parts of OpenMP.

We start with the matrix-matrix multiplication as an easy example to describe our advanced OpenMP syntax. Algorithm 2 shows the new syntax for

Algorithm 2 Matrix multiplication with improved OpenMP

```
1 #pragma omp parallel for schedule(blocked, 64)
2 for (int i=0; i<size; ++i)
3   #pragma omp for schedule(blocked, 64)
4   for (int k=0; k<size; ++k)
5     #pragma omp for schedule(blocked, 64)
6     for (int j=0; j<size; ++j)
7       #pragma omp block
8       C[i][j] += A[i][k] * B[k][j];
```

the matrix multiplication. We introduce both a new scheduling variant called `blocked` and the ability to nest the `#pragma omp for` pragma without using nested parallelism. The OpenMP `for` pragma tells the compiler that the loop iterations can be carried out in any order, which is true for all loops of the example, and the new `blocked` schedule tells the compiler to apply tiling to this loop.

Compilers tile the loops annotated with `schedule(blocked)`. The size of the tiles can be specified as a second parameter of the scheduling-clause or may be automatically determined by a compiler. Having the compiler determine the tile size may not result in the optimal result, but in a reasonable good outcome. Multilevel tiling may be specified by adding not one tile size, but multiple tile sizes to the schedule clause. Explicit tiling can be specified by an additional pragma parameter to identify the variables to which it should be applied. The outer loop, which is generated when tiling is applied to the original loop, remains at the position in the code where the original loop was. The inner loop, in contrast, gets moved directly in front of what we call the *instruction block*. The instruction block should contain only the code that must be executed in every loop iteration. It is identified by `#pragma omp block` and there may only be one instruction block in a tiled loop. In Alg. 2 only line 8 is the instruction block and all newly created loops will be moved in front of this line. When multiple loops are defined as `schedule(blocked)`, the loop order in front of the block is identical to the one of the original loops. It is expected that users make sure that the modified code is still correct, that is e. g. only annotate loops that be tiled. In perfect loop nests where the innermost loop body is the instruction block – as it is the case for the matrix multiplication – this is given if all loop iterations can be carried out in any order. If the loop body consists of more than the instruction block, every code outside the instruction block will only be executed once per tile. We discuss this behavior based on our Gauß-Seidel code next.

The extensions introduced up till now do not allow user to specify dependencies between tiles, as they are used in the Gauß-Seidel example. Listing 3 shows the code with new library functions that overcome this limitation. `omp_num_blocks()` returns the number of tiles a tiled loop is split into, `omp_block_num()` returns the number of the tile currently calculated by the calling thread and `omp_block_size()` returns the size of tile. The functions are

Algorithm 3 Gauß-Seidel with improved OpenMP

```
1  int *counter;
2  omp_lock_t *locks;
3  #pragma omp parallel for schedule(blocked)
4  for (int x=0; x<size; ++x) {
5    #pragma omp single
6    {
7      counter = new int[omp_num_blocks()+1];
8      counter[0] = size;
9      //initialize all other counters with 0
10     locks = new omp_lock_t[omp_num_blocks()+1];
11     //call omp_init_lock for all locks
12   }
13   int y = 0;
14   int x_block = omp_block_num();
15   while (y<size) {
16     omp_set_lock (&locks[x_block]);
17     int lcounter = counter[x_block];
18     omp_unset_lock (&locks[x_block]);
19     #pragma omp for schedule(blocked)
20     for (; y<lcounter; ++y) {
21       #pragma omp block
22       V[y][x] = (V[y][x-1] + V[y][x+1] + V[y-1][x] +
             V[y+1][x])/4;
23       omp_set_lock (&locks[x_block+1]);
24       counter[x_block+1] += omp_block_size();
25       omp_unset_lock (&locks[x_block+1]);
26     }
27   }
28 }
```

always bound to the tiled loop they are directly part of, meaning in our example `omp_block_size()` (Alg. 3, line 24) is bound to the second for-loop. The code to be generated based on Alg. 3 can be found in Alg. 1.

These functions allow users to specify dependencies between tiles. For example in the Gauß-Seidel example one counter and lock per tile in the x-dimension is allocated. In the y-dimension the counter is only updated once per tile, since the update of the code is not part of the instruction-block. To achieve this, first the two created inner loops must be moved in front of the instruction block and the instruction block must be modified so that it no longer uses the old loop indexes but the indexes of the newly created loops. Furthermore the library functions must be created, that is for example `omp_block_num()` must returns the index of the outer tiling loop and `omp_block_size()` the size of the tiles.

The newly suggested `blocked` loop scheduling and the existing `static` scheduling both offer a form of loop tiling. It would have been possible to reuse the `static` scheduler for our modified tiling approach – e. g. the existing behavior is always used when there is no instruction block present. However since, in con-

trast to the existing scheduling variants, `blocked` may influence the correctness of the program, we decided to not reuse the existing name. The extensions play well with all data sharing clauses, however the concept will not ease of tiling in SPMD style OpenMP programs using the `threadprivate` directive for data privatization, since one thread will only execute a subset of tiles and not the whole loop iteration space. We see no way of the extensions would interfere with the existing synchronization concept. Adding the new functions to the runtime system should be rather trivial, as they mostly must only return a value being made available by the tiling.

It is left for future work to analyze the usability of the new extensions for upcoming many-core architectures. However we expect that an user controlled tiling mechanism will be needed for all tile-based many-core architectures, as for example Intel's Single-chip Cloud Computer (SCC) or GPUs. In tile based many-core system it may for example be possible to have a set of closely coupled cores working on a single tile. The new extensions do not tackle the problem of NUMA like remote memory, but concentrate on a way to easily improve cache usage in loop based code. Further research is necessary to identify ways to support NUMA remote memory.

6 Related work

A similar extension for OpenMP has been suggested by Gan et al. [GWMG09], however it offers a subset of the functionality we present. Especially they focus only on perfect loop nests and do not offer direct access to the blocks nor allow using tiles beyond data locality. Compilers like IBMs XLC/C++ and SGI MIPSpro C/C++ offer directive-based support for loop tiling. Extensions for the ZPL [DCS02] and SAC [Sch98] language provide tiling, even though again not with direct access to the tiles. High Performance Fortran (HPF) offers tiling as part of the language. Loop tiling in general has been worked on for several years and discussed in several aspects, e. g. when a compiler can automatically incorporate loop tiling [AMP00]. Optimizations on stencil computation have been analyzed again for several years, a recent study has e. g. been done by Datta et al. [DKW+09].

7 Conclusion / Future work

We demonstrated tiling on two scenarios with the result of increased performance. The main performance increase resulted from the increased data locality of tiling, however tiling also reduced the number of lock and atomic operations. We furthermore demonstrated that using the optimization technique of processor local storage, which is well-known in the GPGPU realm, is beneficial on current CPUs as well. As an addition, we experimented with dynamic tile size in the wavefront pattern to increase the amount of work that can be done in parallel. As a major contribution we showed that these optimizations techniques could be added to OpenMP with only small changes.

Making the notion of tiles available in OpenMP will not only enable developers to specify data locality and thereby increase performance on current CPUs, but lays out the foundation for future work to effectively deploy OpenMP on hardware which natively requires blocks, e. g. GPUs. Further work is required to check if the suggested extensions may result in ambiguous situations, in scenarios different from ones shown in this paper.

References

[AMP00] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 31, Washington, DC, USA, 2000. IEEE Computer Society.

[BFGH07] Michael Bader, Robert Franz, Stephan Günther, and Alexander Heinecke. Hardware-oriented implementation of cache oblivious matrix operations based on space-filling curves. In *PPAM*, volume 4967 of *Lecture Notes in Computer Science*, pages 628–638. Springer, 2007.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[CSG98] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[DCS02] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. High-level language support for user-defined reductions. *J. Supercomput.*, 23(1):23–37, 2002.

[DKW+09] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[GWMG09] Ge Gan, Xu Wang, Joseph Manzano, and Guang R. Gao. Tile Reduction: The First Step towards Tile Aware Parallelization in OpenMP. In *IWOMP '09: Proceedings of the 5th International Workshop on OpenMP*, pages 140–153, Berlin, Heidelberg, 2009. Springer-Verlag.

[McC95] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[Pfi98] Gregory F. Pfister. *In search of clusters (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[Sch98] Sven-Bodo Scholz. On defining application-specific high-level array operations by means of shape-invariant programming facilities. In *APL '98: Proceedings of the APL98 conference on Array processing language*, pages 32–38, New York, NY, USA, 1998. ACM.

[Wol95] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.