University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Tile Reduction: an OpenMP Extension for Tile Aware Parallelization

Ge Gan
Xu Wang
Joseph Manzano
Guang R. Gao

CAPSL Technical Memo 085
December, 2008

Copyright © 2008 CAPSL at the University of Delaware

Email: {gan,wangxu,jmanzano,ggao}@capsl.udel.edu

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

Tiling is widely used by compilers and programmer to optimize scientific and engineering code for better performance. Many parallel programming languages support tile/tiling directly through first-class language constructs or library routines. However, the current OpenMP programming language is *tile oblivious*, although it is the *de facto* standard for writing parallel programs on shared memory systems. In this paper, we introduce *tile aware parallelization* into OpenMP. We propose *tile reduction*, an OpenMP tile aware parallelization technique that allows reduction to be performed on multi-dimensional arrays. The paper has three contributions: **(a)** it is the first paper that proposes and discusses tile aware parallelization in OpenMP. We argue that, it is not only necessary but also possible to have tile aware parallelization in OpenMP; **(b)** the paper introduces the methods used to implement tile reduction, including the required OpenMP API extension and the associated code generation techniques; **(c)** we have applied tile reduction on a set of benchmarks. The experimental results show that tile reduction can make parallelization more natural and flexible. It not only can expose more parallelism in a program, but also can improve its data locality.

1 Introduction

Tiling [1] [2] has been used as an effective compiler optimizing technique to generate high performance scientific codes. Tiling not only can improve data locality for both the sequential and parallel programs [3] , but also can help the compiler to maximize parallelism and minimize synchronization [4] for programs running on parallel machines. Thus, sometimes, it is used by the programmers to hand-tune their scientific programs to get better performance.

Tiling is essentially a program design paradigm. It is a natural representation for many important data objects that are heavily used in scientific and engineering algorithms. Scientific code that is written with the concept of tile/tiling in mind usually looks concise and clear, and thus is much easier to understand and less error prone.

Due to these advantages, it is desirable to provide certain high level language constructs in the programming languages to support tile/tiling in program design directly. To meet this requirement, researchers have proposed various designs in many parallel programming languages or sublanguages. The examples include HPF[5], UPC[6], X10[7], ZPL[8], CAF[9], Titanium[10], and HTA[11], which are among the most popular parallel languages. However, it is interesting to find out that, in the current OpenMP APIs, no directive or clause can be used to annotate data tiles and carry such information to the OpenMP compiler. In other words, the current OpenMP programming language is *tile oblivious*, although it is the *de facto* standard for writing parallel programs on shared memory systems.

In this paper, we propose *tile aware parallelization* for the OpenMP programming language. Its purpose is to enhance the OpenMP API with the concept of tile/tiling so that more data parallelism can be exposed to the OpenMP compiler. Besides granting greater flexibility to the OpenMP compiler to perform more data parallelization, it brings better data locality into the code. This is achieved by extending the current OpenMP directives, clauses, and runtime routines, or introducing new language constructs into OpenMP. Our first effort in this direction is termed *tile reduction*, an OpenMP tile aware parallelization technique that allows parallel reduction to be performed on multi-dimensional arrays.

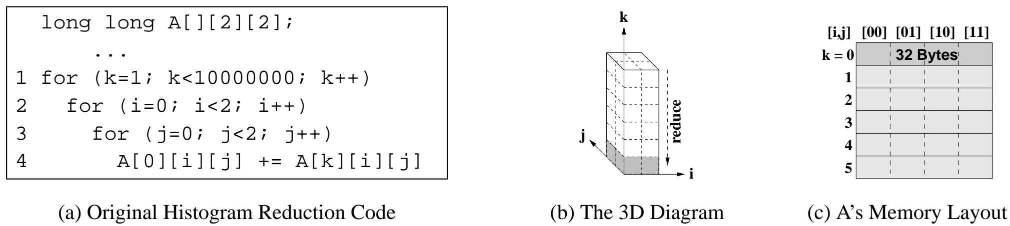
Reduction is a form of recursive calculation that use mathematically associative and commutative operators to ”aggregate” a set of data. Reduction can be performed in parallel to improve performance. For this reason, many programming languages and sub-languages support parallel reduction. Some examples are UPC [12], MPI [13], ZPL [14], and OpenMP [15]. According to the current OpenMP API specification, reduction can only be performed on ”named scalar” variables. It cannot be applied on multi-dimensional arrays. We call this kind of reduction *scalar reduction*. In this paper, we introduce a new technique called *tile reduction*, which evolves the current reduction parallelization from scalar variables to multi-dimensional arrays. We have extended the traditional `reduction` clause to allow the programmers to annotate their code where tile reduction can be applied. We have also developed the required code generation technique to interpret the new `reduction` clause and generate the required parallel code accordingly. The major contributions of this paper are:

1. As far as the authors are aware, this is the first paper that proposes and discusses tile aware parallelization in OpenMP. We argue that, it is not only necessary but also possible to have tile aware parallelization techniques in OpenMP
2. The paper introduces tile reduction, an OpenMP tile aware parallelization technique that applies reduction on multi-dimensional arrays. We discuss the methods used to implement tile reduction, including the required OpenMP API extension and the associated code generation technique.
3. We evaluate the tile reduction technique with a set of benchmarks. The experimental results show that using tile reduction can make the code parallelization more natural and flexible. It not only can expose more parallelism in the program but also can improve its data locality.

The rest of the paper is organized as follows. In Section 2, we use a motivating example to show why tile reduction is necessary. Section 3 will discuss how to implement tile reduction in the OpenMP compiler. We present our experimental data in Section 4 and make our conclusions in Section 5.

2 Motivation

In this section, we use the ”histogram reduction” [16] code as an example to demonstrate the limits of the current OpenMP reduction clause. We will also use the same example to show the advantages of extending *scalar reduction* to *tile reduction*.



(a) Original Histogram Reduction Code (b) The 3D Diagram (c) A’s Memory Layout

Figure 1: The Histogram Reduction Example

Figure 1(a) shows the code of the histogram reduction program. The code works on `A[][][]`, a 3-dimensional array with each element containing an 8-byte `long long`. It aggregates all elements along the `k` dimension and stores the results in the `2x2` tile `A[0][][]`. The diagram in Figure 1(b) shows these operations. We assume that the cache line size is 32 bytes and that the the array is stored in a row-major order in the memory. Therefore, elements with the same `k` coordinate can be fed into the same cache line, as shown in Figure 1(c). There are three nested loops in the code. Each loop traverses one of the `i`, `j`, `k` dimension of the array. Data dependence only exit in loop `k` because of the recursive calculation.

```
0 for (k=1; k<10000000; k++)
1   #pragma omp parallel for
2   for (i=0; i<2; i++)
3     for (j=0; j<2; j++)
4       A[0][i][j] += A[k][i][j]
```

(a) Parallelize loop "i"

```
0 for (k=1; k<10000000; k++)
1   #pragma omp parallel for collapse(2)
2   for (i=0; i<2; i++)
3     for (j=0; j<2; j++)
4       A[0][i][j] += A[k][i][j]
```

(b) Parallelize loop "i" and "j" using the collapse clause

Figure 2: Parallelize the Histogram Reduction Program Without Changing the Code

Given the code in Figure 1(a), there are many different ways to parallelize it. However, due to the data dependence in loop `k`, we cannot parallelize this loop. Therefore, without changing the code, we can only parallelize loop `i` and `j`, as shown in Figure 2(a) and 2(b). It is obvious that there are trivial workload and little parallelism in loop `i` and loop `j`. Thus, it is not worthwhile to parallelize these two loops, even while using the `collapse` clause (supported in OpenMP 3.0 [15]).

```
0 #pragma omp parallel for
1 for (j=0; j<2; j++)
2   for (i=0; i<2; i++)
3     for (k=1; k<10000000; k++)
4       A[0][j][i] += A[k][j][i]
```

(a) Parallelize the outer loop

```
0 #pragma omp parallel for collapse(2)
1 for (j=0; j<2; j++)
2   for (i=0; i<2; i++)
3     for (k=1; k<10000000; k++)
4       A[0][j][i] += A[k][j][i]
```

(b) Parallelize the outer two loops

```
0 #pragma omp parallel for private(sum) collapse(2)
1 for (j=0; j<2; j++)
2   for (i=0; i<2; i++) {
3     sum = 0;
4 #pragma omp parallel for shared(sum) reduction(+:sum)
5     for (k=0; k<10000000; k++)
6       sum += A[k][j][i]
7     A[0][j][i] = sum;
8   }
```

(c) Nested parallelization to harvest more parallelism

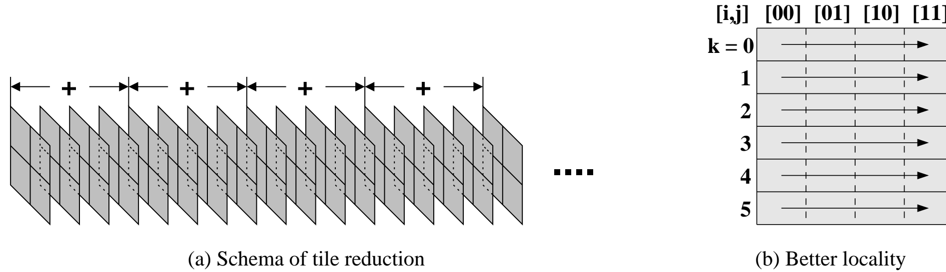
[i,j] **[00]** **[01]** **[10]** **[11]**
k = 0, **1**, **2**, **3**, **4**, **5**

(d) Data access pattern

Figure 3: More Parallelization for Histogram Reduction Code

To get a larger workload and more parallelism, we can interchange the loops manually before parallelizing the code, as shown in Figure 3. In Figure 3(a) and 3(b), the workload that can be assigned to the threads is large enough. However, the available parallelism is still very small (only supports two

or four concurrent threads). Figure 3(c) shows a better solution. In Figure 3(c), a nested `parallel for` directive is used to parallelize the recursive addition using the `reduction` clause (with trivial code change). Although the code in Figure 3(c) can leverage all levels of parallelism in the program, its strided data access pattern would cause a great number of unnecessary cache misses, as shown in Figure 3(d). Code in Figure 3(a) and 3(b) have the same data locality problem. Apparently, the current OpenMP parallelization techniques cannot harvest the maximum parallelism and data locality in the code at the same time. They suffer from either insufficient parallelism or poor data locality.



(a) Schema of tile reduction (b) Better locality

Figure 4: The Ideal Parallelization Schema for the Histogram Reduction Code

The ideal parallelization is shown in Figure 4. Logically, the recursive addition can be viewed as being performed on an array of `2x2` data tiles. In theory, these tiles can be added together in parallel by multiple threads, as shown in Figure 4(a). In this way, the code can achieve both the maximum parallelism and the best data locality (see Figure 4(b)). Besides, from the programmers' angle, this is the most natural way to perform parallelization on this piece of code. However, the current OpenMP specification does not provide any mechanism to support such kind of parallelization. This motivates us to extend the traditional *scalar* reduction to *tile* reduction.

3 Tile Reduction

In this section, we will discuss the techniques used to implement tile reduction. They include the extended OpenMP programming interface and the required code generation design. The related runtime support will be mentioned when needed.

3.1 Programming Interface Extension

In order to support tile reduction, we need to extend the current OpenMP programming interface. The extension was made based on three criteria. First, it must be able to cover most of the common cases of tile reduction code. Second, it must be simple and easy to use and provide the programmers with the maximal flexibility. Third, the extension should not complicate the code generation of the OpenMP compiler and the OpenMP runtime. Figure 5(a) shows the OpenMP API (C/C++) extension we proposed for the `reduction` clause. Figure 5(b) gives a simple example that uses the extended `reduction` clause to parallelize the tile reduction code.

```
reduction(operator : T[j_k, L_k, U_k]...[j_2, L_2, U_2][j_1, L_1, U_1])
```

T: Tile name
k: Dimension of the tile
j_i: the loop index that is used in the traversal of the ith dimension of the tile
L_i: the lower bound of j_i
U_i: the strict upper bound of j_i

(a) OpenMP API (C/C++) extension for the `reduction` clause

```
  int B[2][2] = {{0,0},{0,0}};
          ...
0 #pragma omp parallel for reduction(+: B[j,0,2][i,0,2])
1 for (k=0; k<10000000; k++)
2   for (j=0; j<2; j++)
3     for (i=0; i<2; i++)
4       B[j][i] += A[k][j][i]
```

(b) Simple example using the extended API

Figure 5: OpenMP API (C/C++) extension and a simple example code

Compared with the current OpenMP API specification, the difference is in the `list` construct. In addition to the ”named scalar” variables, we allow the programmers to put a ”multi-dimensional array” in the `list` construct. This ”multi-dimensional array” is not a real array data structure in the language sense. It is a language construct that conveys important information to the OpenMP compiler. It tells the compiler the shape, the size, and the element type of the tile and how its elements are traversed by the loops.

To make the paper easy to follow, we call the tile under reduction as the *reduction tile*; the ”multi-dimensional array” in the `list` construct as the *tile descriptor*; and the loops involved in performing ”one” recursive calculation as the *reduction kernel loops*. For the example in Figure 5(b), the reduction tile is `B[][]`, the tile descriptor is `B[j,0,2][i,0,2]`, and the reduction kernel loops are the `j` and `i` loops (not including the `k` loop, i.e., the parallelized loop). In our design, the shape of the reduction tile must be a rectangle or a high-dimensional rectangle. Triangle or other shapes are not yet supported. The exact shape and size of the reduction tile are determined by the tile descriptor.

The format of the tile descriptor is shown in Figure 5(a). It has two parts: the *tile name* (i.e., `T`) and the *dimension descriptor* (i.e., $[j_k, L_k, U_k]...[j_2, L_2, U_2][j_1, L_1, U_1]$). Tile name must be the same as the multi-dimensional array variable on which the recursive calculations are performed. For the example in Figure 5(b), this corresponds to the name of the *lhs* variable in line 4, which is `B`. It tells the OpenMP compiler the data type of the tile element, which must be a built-in scalar type. The dimension descriptor, on the other hand, is an array of 3-tuples. Each 3-tuple corresponds to one dimension of the tile and stores important information of that dimension. These 3-tuples are listed in the dimension descriptor in descendant order (higher dimension first). Each 3-tuple has three elements: loop index variable, upper bound expression, and lower bound expression. The loop index variable identifies a loop in the reduction kernel loops. Since stride accesses are not allowed, the loop stride is always 1, so it is

omitted from the tuple. The size of the k-dimensional tile is calculated from equation (1).

$$(U_k - L_k) \times ...(U_2 - L_2) \times (U_1 - L_1) \tag{1}$$

The information stored in the tile descriptor is very important for the OpenMP compiler to generate correct parallel code.

The `operator`, as usual, must be a mathematically associative and commutative operator that performs the recursive calculation. In our current example, it is a "+".

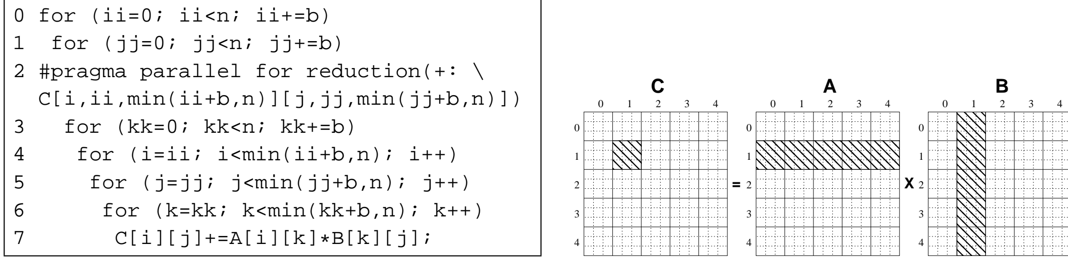
```
0 #pragma omp parallel for reduction(+: A[j,0,2][i,0,2])
1 for (k=1; k<10000000; k++)
2  for (j=0; j<2; j++)
3   for (i=0; i<2; i++)
4     A[0][j][i] += A[k][j][i]
```

Figure 6: Tile reduction: tile is part of a bigger multi-dimensional array

The reduction tile is not required to be a standalone multi-dimensional array. Instead, it can be part of another larger multi-dimensional array. For example, in the code in Figure 6, the reduction tile is `A[0][j][i]` ($j = \{0,1\}, i = \{0,1\}$). It is a 2×2 slice cut out from the 3-dimensional array `A[][][]`;

Besides, as we have mentioned before, the lower and upper bounds in the dimension descriptor are expressions. They are not required to be constants. Generally, the lower and upper bounds can be a function of other variables, as long as the result of the function can be decided at runtime. Figure 7 shows such an example. The code in Figure 7 is a blocked matrix multiplication program. It is easy to see that there is an opportunity to apply tile reduction on the loop in line 3, i.e., the `kk` loop. The diagram on the right hand side gives an intuitive illustration. In this example, the reduction tiles are blocks cut out from a big 2×2 matrix (`C[][]`). Therefore, the lower and upper bounds of the reduction tiles are not fixed values. In addition, the matrix `C[][]` might not be able to be evenly blocked. So, the tiles located at the margin of the matrix are usually smaller than the tiles located inside of the matrix. Thus, the sizes of the reduction tiles are not necessarily the same. All these information is reflected in the lower and upper bound expressions (or functions) in the dimension descriptor. Moreover, there is a restriction for the lower bound and upper bound expressions. They should not be functions of any index variable in the reduction kernel loops, i.e., they are orthogonal. This is to make sure that the shape of the reduction tile is a rectangle, or high-dimensional rectangle.

An interesting observation of this example code is that the number of the reduction kernel loops (which is 3, from line 4 to line 6) is not the same as the dimension of the reduction tile (which is 2). Generally, we do not require the number of the reduction kernel loops to be the same as the dimension of the reduction tile. We only require that the operations performed by the code in the reduction kernel loops can be viewed as one associative and commutative *macro* operation performed on the entire reduction tile.



```
0 for (ii=0; ii<n; ii+=b)
1  for (jj=0; jj<n; jj+=b)
2 #pragma parallel for reduction(+: \
  C[i,ii,min(ii+b,n)][j,jj,min(jj+b,n)])
3   for (kk=0; kk<n; kk+=b)
4    for (i=ii; i<min(ii+b,n); i++)
5     for (j=jj; j<min(jj+b,n); j++)
6      for (k=kk; k<min(kk+b,n); k++)
7       C[i][j]+=A[i][k]*B[k][j];
```


Figure 7: Tile reduction: upper and lower bounds are functions

3.2 Code Generation

Since tile reduction is derived from scalar reduction, its code generation shares the same framework as scalar reduction. Thus, we illustrate the code generation for tile reduction under the same framework as scalar reduction and use the code generation for scalar reduction as a reference. Generally, the code generation needs to deal with the following problems:

1. Distribute the iterations of the parallelized loop among the threads;

2. Allocate memory for the private copy of the tile used in the local recursive calculation;

3. Perform the local recursive calculation which is specified by the reduction kernel loops;

4. Update the global copy of the reduction tile;

Figure 8 shows the code generated for the tile reduction example in Figure 7. To make the paper easy to follow, we present the pseudo C code in the figure.

As we have mentioned at the beginning of Section 3.1, we try to avoid complicating the code generation when we were developing the extension for the `reduction` clause. A good example is the code generation for distributing the iterations of the parallelized loop among the dynamic threads. Actually, this part of the code generation for tile reduction is the same as that for scalar reduction.

In the tile reduction program, the reduction kernel loops can be viewed as a single statement that performs the recursive calculation, which is the same as its counterpart in the scalar reduction program. So, from the angle of iteration distribution, the scalar reduction code and the tile reduction code are logically the same. Therefore, the method used to generate iteration distribution code for scalar reduction can also be used to generate iteration distribution code for tile reduction. It doesn't matter which `schedule` policy (`static`, `dynamic`, `guided`, or `runtime`) is deployed.

In Figure 8, we use the `static` scheduling policy as an example. In the code from line 2 to line 6, the iterations of the `kk` loop (line 3 in Figure 7) are evenly distributed among the threads. The iterations of the loop are divided into chunks and each chunk is assigned to one dynamic thread. The iteration chunk assigned to the thread is delimited by the lower bound variable `"lb"` and the upper bound variable `"ub"`, which are determined by the *thread number* of the owner thread. This piece of

```
0
1  /* statically partition the iteration space among the threads */
2  num_thr = __builtin_omp_get_num_threads ();
3  thr_id = __builtin_omp_get_thread_num ();
4  chunk_size = (((n+(b-1))/(b-1))%num_thr) == 0 ? \
     (((n+(b-1))/(b-1))/num_thr) : (((n+(b-1))/(b-1))/num_thr)+1;
5  lb = chunk_size * thr_id; /* lower bound */
6  ub = min((lb+chunk_size),n); /* upper bound */
7
8  /* allocate memory for private tile */
9  private_tile = (int *)__builtin_omp_memory_alloc( \
                    (min(ii+b,n)-ii)*(min(jj+b,n)-jj)*sizeof(int));
10
11  /* local tile reduction: serial */
12  for (kk=lb; kk<ub; kk+=b)
13   for (i=ii; i<min(ii+b,n), i++)
14     for (j=jj; j<min(jj+b,n), j++)
15       for (k=kk; k<min(kk+b,n), kk++)
16         private_tile[i-ii][j-jj] += A[i][k]*B[k][j]
17
18  /* update the global reduction tile */
19  __builtin_omp_atomic_start ();
20  for (i=ii; i<min(ii+b,n), i++)
21    for (j=jj; j<min(jj+b,n), j++)
22      C[i][j] += private_tile[i-ii][j-jj];
23  __builtin_omp_atomic_end ();
24
25  free(private_tile);
26
```

Figure 8: Pseudo code generated for the matrix multiplication example to perform tile reduction

code only deals with the parallelized loop and the user specified OpenMP parameters. It does not even need to look into the code of the reduction kernel loops. This is the same for other schedule policies.

In line 9, the OpenMP runtime routine allocates memory for the the private tile (`private_tile`), which is a 2-dimensional array. This private tile is used by the thread as a temporary storage to perform the local sequential tile reduction. Its size is calculated from the parameters specified in the dimension descriptor (see equation 1). Its element data type is inferred from the tile name. All this information is obtained from the extended `reduction` clause.

The local sequential tile reduction is performed in the code from line 12 to line 16. This piece of code is almost the same copy as the original sequential program (line 3 to line 7 in Figure 7) except two places. In line 12, the lower and upper bounds of the loop are changed to `"lb"` and `"ub"`. This is to restrict the range of the iteration space in the chunk assigned to the current thread. Besides, in line 16, we replace the original reduction tile with the private tile and update its indices. This index calibration is required because the global reduction tile is cut out from a bigger multi-dimensional array, while the private tile is a standalone array. This piece of code performs local tile reduction sequentially, as in the original un-parallelized code.

After finishing the local tile reduction, the thread must update the global reduction tile. The code is shown in line 19 to line 23. The runtime routines in lines 19 & 23 ensure atomic access to the global

reduction tile. The loops in line 20 and line 21 are extracted from the *reduction kernel loops*. Only the loops listed in the *tile descriptor* are selected. So, the loop `k` in the reduction kernel loops is not included. The *lhs* variable of the statement in line 22 is the same variable as in the original code (line 7 in Figure 7). However, the *rhs* variable has been replaced with the private tile, in which the indices have been updated.

From the code in Figure 8, it is easy to see that the code generation for the tile reduction is as easy as that for the traditional scalar reduction. Meanwhile, no extra runtime supports is required. These advantages make the implementation of tile reduction in the OpenMP compiler very easy. In the next section, we will present the experimental results of applying the tile reduction on several typical benchmarks.

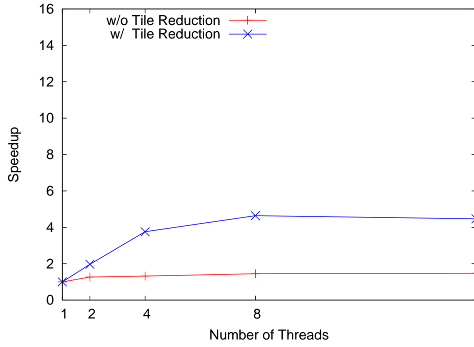
4 Experiments

We have applied tile reduction on three benchmarks: the 2D histogram reduction, matrix-matrix multiplication and matrix-vector multiplication. The required code generation was implemented through source-to-source transformation and was prototyped in the Omni-1.6 OpenMP compiler [17]. The machine used in the experiments has 4 Intel Dual-Core Xeon (Paxville) chips, which are clocked at 3.0 GHz. Each core has HyperThreading (HT) enabled. Therefore, the machine can be viewed as a 16-processor shared memory parallel computer. Each chip has 4MB L2 cache (2MB each core) and each core has 16KB L1 cache.

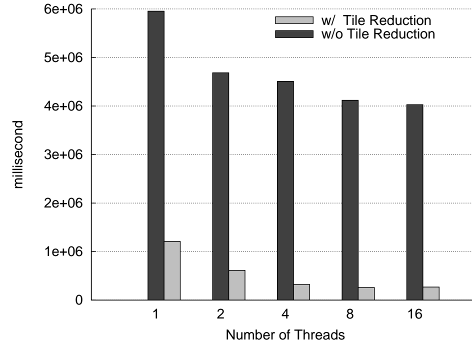
Figure 9 shows the experimental data of the three benchmarks. The curve graphs on the left column display the speedup of the benchmark programs parallelized either through the tile reduction clause (w/ tile reduction) or through the standard OpenMP APIs (w/o tile reduction). The bar charts on the right column demonstrate the difference of the absolute execution time between the corresponding programs (w/ and w/o tile reduction) of the same set of benchmarks.

Figure 9(b) shows great performance enhancement if we parallelize the 2D histogram reduction benchmark with the tile reduction clause. Generally, compared with the program parallelized with standard OpenMP pragma, the absolute execution time of the tile reduction version decreased about 90% and its speedup on 8 threads increased from 1.5 to 4.5. The performance gain comes from the improved data locality, which owes to the tile reduction optimization. Without using tile reduction, the 2D histogram reduction program exhibit very poor scalability (shown in Figure 3). The tile reduction parallelization successfully rectifies the data access pattern and thus significantly improves its scalability. However, no matter what kind of optimizations are used, this benchmark stops scaling beyond 8 threads. This is because of the huge number of memory references in the code, which results in that its performance is finally restricted by the bandwidth of the shared memory bus.

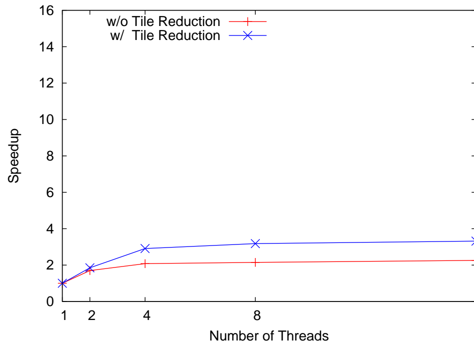
The same phenomena are also observed in the matrix-matrix multiplication benchmark (see Figure 9(c) and 9(d)). Tile reduction can also decrease its execution time and improve its scalability. However, the magnitude of the performance enhancement caused by tile reduction is not as big as that of the 2D histogram reduction benchmark. This is also the same for the scalability enhancement. The reason is that the data locality of the tiled matrix-matrix multiplication program is better than the 2D histogram



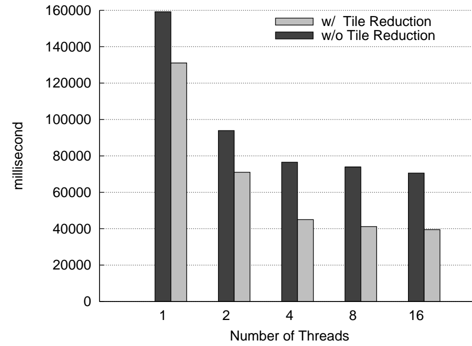
(a) 2D histogram reduction: speedup



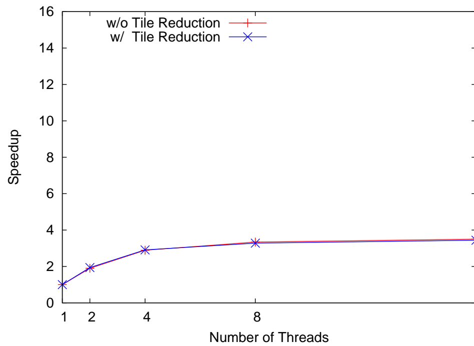
(b) 2D histogram reduction: execution time



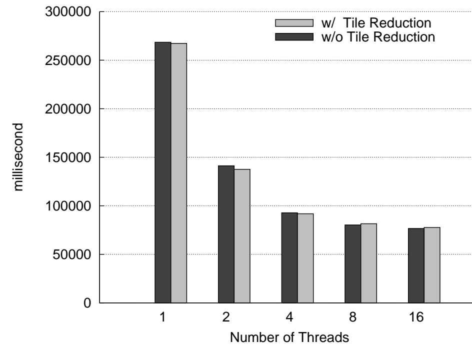
(c) Matrix-matrix multiplication: speedup



(d) Matrix-matrix multiplication: execution time



(e) Matrix-vector multiplication: speedup



(f) Matrix-vector multiplication: execution time

Figure 9: Comparison of the speedup and execution time between the program parallelized with tile reduction and the program parallelized with the standard OpenMP pragma.

reduction benchmark. Therefore, the performance gain from tile reduction in the matrix multiplication program is less than that in the 2D histogram reduction program. On average, the execution time decreased 34% after applying tile reduction and its speedup increased from 2.15 to 3.18 on 8 threads and from 2.26 to 3.32 on 16 threads.

For the matrix-vector multiplication case, the performance enhancement brought about by tile re-

duction is smaller than that of the previous two benchmarks. The reason is the same as the previous one. Moreover, compared with the other two benchmarks, there are less data memory references in this benchmark. So, the program's performance degrades a little bit when it runs with 8 or 16 threads. This is because of the synchronization overhead caused by the code in line 19 and 23 in Figure 8. In average, its execution time decreased 0.28%.

5 Summary and Conclusions

In this paper, we introduced the concept of tile aware parallelization for OpenMP. Meanwhile, we developed the first tile aware parallelization technique - tile reduction, and illustrated the details of code generation for the tile reduction clause. We also designed a series of experiments to evaluate the tile reduction technique. From the experimental results and our experience of parallelizing the benchmarks, we have the following conclusions:

1. As a building block of the tile aware parallelization theory, tile reduction brings more opportunities to parallelize dense matrix applications.
2. For some benchmarks, tile aware parallelization is a more natural and intuitive way to reason about the best parallelization decision.
3. Tile reduction not only can improve data locality for some programs, but also can expose more parallelism.

6 Related Work

Parallel reduction operations are supported in many parallel programming languages. They include C**[18], SAC [19], ZPL [16], UPC [12], and MPI [13]. Most of them support user-defined reduction operations, either through language constructs or through library routines. User-defined reduction operation provides a flexible way to implement tile reduction. However, programmers need to change both data structures and algorithms, which, sometimes, is not a tirivial job.

Another piece of work that we need to mention is [20]. In [20], the authors propose to extend the OpenMP `reduction` clause to parallelize C++ generic algorithms. They propose to support user-defined types, overloaded operators, and function objects in the same way as the built-ins supported in the current OpenMP `reduction` clause. Their work is very close to that presented in this paper. However, we study the reduction problem from a different angle. We propose tile reduction as one of the tile aware parallelizing technique for OpenMP, while [20] proposes user-defined reduction operation to complete their OpenMP extensions for parallelizing generic libraries. In our tile aware parallelization technique, we are concerned with the data partition, locality and a more flexible and efficient way to parallelize dense matrix programs written in cannonical C syntax, while the purpose of [20] is to allow people to parallelize programs written in modern C++ idioms such as *iterators* and *function objects*, which are not cannonical C syntax. Second, due to the non-trivial dynamic overhead of the generic

techniques, generic libraries are not widely used in programming high performance scientific and engineering algorithms. Finally, there are no experimental data in [20].

7 Future Work

Tile reduction is one of the building block of the tile aware paralleization technique developed for OpenMP. One of our future work is to develop more parallelizing techniques (like tile reduction) such that OpenMP compiler can ”recognize” data tiles and allow its runtime library to manipulate them. Our goal is to add tile aware parallelizing directives or clauses into the OpenMP programming interface. The purpose is to evolve OpenMP into an appropriate programming model for many-core processors with explicitly managed memory hierarchy [21], e.g. the IBM CELL [22] and the IBM Cyclops-64 [23] processor.

Acknowledgments

This work was supported by NSF (CNS-0509332, CSR-0720531, CCF-0833166, CCF-0702244), and other government sponsors. We thank all the members of CAPSL group at University of Delaware. We thank Jason Lin and Lei Huang for their valuable comments and feedback.

References

[1] Anderson, J.M., Amarasinghe, S.P., Lam, M.S.: Data and computation transformations for multiprocessors. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, Santa Barbara, California (July 19–21, 1995) 166–178 *SIGPLAN Notices,* 30(8), August 1995.

[2] Anderson, J.M., Lam, M.S.: Global optimizations for parallelism and locality on scalable parallel machines. In: Proceedings of the ACM SIGPLAN ’93 Conference on Programming Language Design and Implementation, Albuquerque, New Mexico (June 23–25, 1993) 112–125 *SIGPLAN Notices,* 28(6), June 1993.

[3] Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: Proceedings of the ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation, Toronto, Ontario (June 26–28, 1991) 30–44 *SIGPLAN Notices,* 26(6), June 1991.

[4] Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: Conference Record of POPL’97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris (January 15–17, 1997) 201–214

[5] High Performance Fortran Forum: High-performance fortran language specification version 2.0. Technical report, Rice University (1997)

[6] El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K.: UPC: Distributed Shared-Memory Programming. Wiley-Interscience (2003)

[7] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2005) 519–538

[8] Deitz, S.J.: High-level programming language abstractions for advanced and dynamic parallel computations. PhD thesis, Seattle, WA, USA (2005) Chair-Lawrence Snyder.

[9] Dotsenko, Y., Coarfa, C., Mellor-Crummey, J.: A multi-platform co-array fortran compiler. In: PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, IEEE Computer Society (2004) 29–40

[10] Hilfinger, P.N., Bonachea, D., Gay, D., Graham, S., Liblit, B., Pike, G., Yelick, K.: Titanium language reference manual. Technical report, Berkeley, CA, USA (2001)

[11] Guo, J., Bikshandi, G., Fraguela, B.B., Garzaran, M.J., Padua, D.: Programming with tiles. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2008) 111–122

[12] UPC Consortium: UPC Collective Operations Specifications V1.0 A publication of the UPC Consortium (2003)

[13] Forum, M.P.I.: MPI: A message-passing interface standard (version 1.0). Technical report (May 1994) URL http://www.mcs.anl.gov/mpi/mpi-report.ps.

[14] Deitz, S.J., Chamberlain, B.L., Choi, S.E., Snyder, L.: The design and implementation of a parallel array operator for the arbitrary remapping of data. In: PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2003) 155–166

[15] OpenMP Architecture Review Board: OpenMP Application Program Interface Version 3.0 (May 2008) http://www.openmp.org/mp-documents/spec30.pdf.

[16] Deitz, S.J., Chamberlain, B.L., Snyder, L.: High-level language support for user-defined reductions. J. Supercomput. **23**(1) (2002) 23–37

[17] Kusano, K., Satoh, S., Sato, M.: Performance evaluation of the omni openmp compiler. In: ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing, London, UK, Springer-Verlag (2000) 403–414

[18] Viswanathan, G., Larus, J.R.: User-defined reductions for efficient communication in data-parallel languages. Technical Report 1293, University of Wisconsin-Madison (Jan 1996)

[19] Scholz, S.B.: 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, New York, NY, USA, ACM (1998) 32–38

[20] Kambadur, P., Gregor, D., Lumsdaine, A.: Openmp extensions for generic libraries. In: Lecture Notes in Computer Science: OpenMP in a New Era of Parallelism, IWOMP'08, International Workshop on OpenMP. Volume 5004/2008., Springer Berlin / Heidelberg (2008) 123–133

[21] Knight, T.J., Park, J.Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P.: Compilation for explicitly managed memory hierarchies. In: PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2007) 226–236

[22] Eichenberger, A.E., O'Brien, K., O'Brien, K., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M.: Optimizing compiler for the cell processor. In: PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, IEEE Computer Society (2005) 161–172

[23] del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. In: Workshop on Modeling, Benchmarking and Simulation (MoBS'05) of ISCA'05, Madison, Wisconsin (June 2005)