Tile Percolation: an OpenMP Tile Aware Parallelization Technique for the Cyclops-64 Multicore Processor

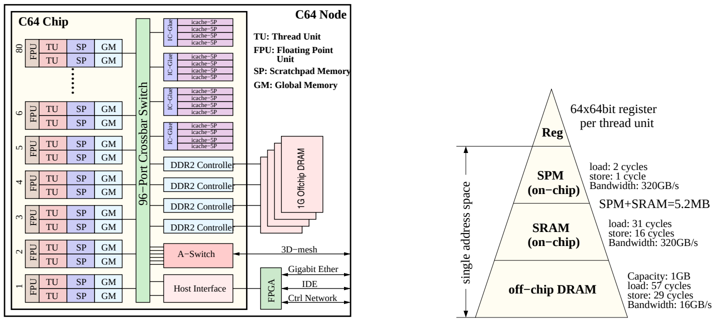
Ge Gan Xu Wang Joseph Manzano Guang R. Gao

Department of Electrical and Computer Engineering
University of Delaware
Newark, Delaware 19716, U.S.A
{gan,wangxu,jmanzano,ggao}@capsl.udel.edu

Abstract. Programming a multicore processor is difficult. It is even more difficult if the processor has software-managed memory hierarchy, e.g. the IBM Cyclops-64 (C64). A widely accepted parallel programming solution for multicore processor is OpenMP. Currently, all OpenMP directives are only used to decompose computation code (such as loop iterations, tasks, code sections, etc.). None of them can be used to control data movement, which is crucial for the C64 performance. In this paper, we propose a technique called *tile percolation*. This method provides the programmer with a set of OpenMP pragma directives. The programmer can use these directives to annotate their program to specify *where* and *how* to perform data movement. The compiler will then generate the required code accordingly. Our method is a semi-automatic code generation approach intended to simplify a programmer's work. The paper provides **(a)** an exploration of the possibility of developing pragma directives for semi-automatic data movement code generation in OpenMP; **(b)** an introduction of techniques used to implement tile percolation including the programming API, the code generation in compiler, and the required runtime support routines; **(c)** and an evaluation of tile percolation with a set of benchmarks. Our experimental results show that tile percolation can make the OpenMP programs run on the C64 chip more efficiently.

1 Introduction

OpenMP [1] is the *de facto* standard for writing parallel programs on shared memory multiprocessor system. For the IBM Cyclops-64 (C64) processor [2, 3], OpenMP is one of the top selected programming model. As shown in Figure 1(a), the C64 chip has 160 homogeneous processing cores. They are connected by a 96-port, 7-stage, non-blocking on-chip crossbar switch [4]. The chip has 512KB instruction cache but no data cache. Instead, each core contains a small amount of SRAM (5.2MB in total) that can be configured into either Scratchpad Memory (SPM), or Global Memory (GM), or both in combination. Off-chip DRAM are attached onto the crossbar switch through 4 on-chip DRAM controllers. All memory modules are in the same address space and can be accessed directly by all processing cores [5]. However, different segment of the memory address space has different access latency and bandwidth. See Figure 1(b) for the detailed parameters of the C64 memory hierarchy. Roughly speaking, the C64 chip is a single-chip shared memory multiprocessor system. Therefore, it is easy to land



(a) C64 Architecture

(b) C64 Memory Hierarchy

Fig. 1. The IBM Cyclops-64 Multicore Processor

OpenMP on the C64 chip [6]. However, due to its *software-managed* memory hierarchy, making an OpenMP program run efficiently on the C64 chip is not a trivial task.

Given a processor like C64, it is very important for the OpenMP program to fully utilize the on-chip memory resources. This requires the programmer to insert code in the program to move data back and forth explicitly between the on-chip and off-chip memory. Thus, the program can benefit from the short latencies of the on-chip memory and the huge on-chip bandwidth. Unfortunately, this would make the C64 multicore processor more difficult to program. From the OpenMP methodology, we have learned that it would be very helpful if we could annotate the program with a set of OpenMP pragma directives to specify where data movement is beneficial and possible, and let the compiler generate the required code accordingly. This is just like using the `parallel for` directive to annotate a loop and let the OpenMP compiler generate the multi-threaded code. This would free the programmer from writing tedious data movement code.

In this paper, we introduce *tile percolation*, a tile aware parallelization technique [7] for the OpenMP programming model. The philosophy behind the tile aware parallelization technique is to enable OpenMP programmers not only the capability to direct the compiler to perform computation decomposition, but also the power to direct the compiler to perform data movement. The programmer will be provided with a set of simple OpenMP pragma directives. They can use these directives to annotate their program to instruct the compiler *where* and *how* data movement will be performed. The compiler will generate the correct computation and data movement code based on these annotations. At runtime, a set of routines will be provided to perform the dynamic data movement operations. This not only makes the programming on the C64 chip easier, but also makes sure that the data movement code inserted into the program is optimized. Since the major data objects being moved are ”sub-blocks” in the multi-dimensional array, this approach is termed *tile percolation*. The major contributions of the paper are as follows:

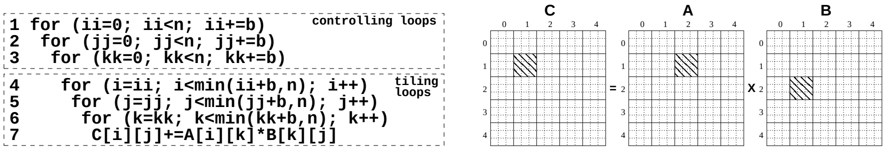
1. As far as the authors are aware, this is the first paper that explores the possibility of using pragma directives for semi-automatic data movement code generation in OpenMP.

2. The paper has introduced the techniques used to implement tile percolation including the programming API, the code generation in compiler, and the required runtime support.
3. We have evaluated tile percolation with a set of benchmarks. Our experimental results show that tile percolation can make the OpenMP programs run on the C64 chip more efficiently.

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

2 Motivation

In this section, we use the tiled matrix multiplication code as a motivating example to demonstrate why writing efficient OpenMP program for the software-managed C64 memory hierarchy is not trivial and why a semi-automatic code generation approach is necessary.



```
for (ii=0; ii<n; ii+=b)
 for (jj=0; jj<n; jj+=b)
  for (kk=0; kk<n; kk+=b)
   for (i=ii; i<min(ii+b,n); i++)
    for (j=jj; j<min(jj+b,n); j++)
     for (k=kk; k<min(kk+b,n); k++)
      C[i][j]+=A[i][k]*B[k][j]
```

(a) Tiled Matrix Multiplication Code

(b) Data Access Pattern in the Tiling Loops

Fig. 2. Tiled Matrix Multiplication: C = A x B

Figure 2 shows the tiled matrix multiplication code [1] and the data access pattern of the kernel loops. On the C64 chip, to make sure that this program fully utilizes the on-chip memory resources, the programmers need to insert tile movement code manually in the source code to move data tiles back and forth between the on-chip and off-chip memory. Figure 3 shows the examples of the manually inserted code. In both examples, no matter how the computations in the *controlling loops* are decomposed among the cores, for the tiling loops, small data tiles are moved into the on-chip SRAM memory and the associated computations are performed there. Figure 3(a) shows the naive version, in which the array elements are copied into the on-chip memory one by one. A better version is shown in Figure 3(b), which utilizes the off-chip memory bandwidth more efficiently. In both versions, the programmers need to study the original

[1] Because of the advantages of the *surface-to-volume* effect [8], the *algorithm-by-tile* approach [9, 10] is used intensively in developing scientific and engineering code. For instance, the LAPACT programs [11, 12] use many level-3 BLAS code [13] to leverage the computer's memory hierarchy, no matter if the memory hierarchy is managed by hardware or software, or if it is managed implicitly or explicitly.

```
/* allocate on-chip memory */
AA=(float *)sram_malloc(...);
BB=(float *)sram_malloc(...);
CC=(float *)sram_malloc(...);
      ...
/* item-by-item memory copy */
for (i=ii;i<min(ii+b,n);i++)
 for (j=jj;j<min(jj+b,n);j++)
  for (k=kk;k<min(kk+b,n);k++){
    AA[i-ii][k-kk] = A[i][k];
    BB[k-kk][j-jj] = B[k][j];
    CC[i-ii][j-jj] = C[i][j];
  }
/* MxM performed on-chip */
for (i=0;i<min(b,n-ii);i++)
 for (j=0;j<min(b,n-jj);j++)
  for (k=0;k<min(b,n-kk);k++)
   CC[i][j]+=AA[i][k]*BB[k][j];
/* copy out the results */
for (i=ii;i<min(ii+b,n);i++)
 for (j=jj;j<min(jj+b,n);j++)
  for (k=kk;k<min(kk+b,n);k++)
    C[i][j]=CC[i-ii][j-jj];
```

(a) A naive version

```
/* allocate on-chip memory */
AA=(float *)sram_malloc(...);
BB=(float *)sram_malloc(...);
CC=(float *)sram_malloc(...);
      ...
/* mcpy: optimized memory copy routine */
for (i=ii; i<min(ii+b,n); i++)
 mcpy(&CC[i-ii][0],&C[i][jj],min(b,n-jj))
for (k=kk; k<min(kk+b,n); k++)
 mcpy(&BB[k-kk][0],&C[k][jj],min(b,n-jj))
for (i=ii; i<min(ii+b,n); i++)
 mcpy(&AA[i-ii][0],&A[i][kk],min(b,n-kk))
           ...
/* on-chip calculation */
for (i=0; i<min(b,n-ii); i++)
 for (j=0; j<min(b,n-jj); j++)
  for (k=0; k<min(b,n-kk); k++)
    CC[i][j]+=AA[i][k]*BB[k][j];
              ...
/* copy out the results */
for (i=ii; i<min(ii+b,n); i++)
 mcpy(&C[i][jj],&CC[i-ii][0],min(b,n-jj))
              ...
```

(b) An optimized version

Fig. 3. Examples of Manually Inserted Data Movement Code (Pseudo Code)

source code carefully to figure out how to write correct and efficient data movement code. They are forced to deal with the convoluted array index calculation, which makes their work more complicated.

A simpler approach is to let the compiler to generate the required data movement code automatically. In [14, 15], the authors present their implementation of this idea on the IBM CELL processor. However, in [16], it is revealed that the performance of the automatically generated code is worse than the performance of the manually reformed code [2]. The reason is not because the compiler can not generate the optimal code, but because the static analysis performed by the compiler is not powerful enough to capture all the beneficial cases (which is a well-told story). This motivates us to develop a novel semi-automatic approach: the programmer specifies the place in which efficient data movement is crucial, while the compiler, accordingly, generates the required high quality code.

3 Tile Percolation

In this section, we will use a simple example to demonstrate how to implement tile percolation. It includes three parts: the programming API, the data movement code generation in the compiler, and the required runtime support.

3.1 Programming API

In the design of the programming API for tile percolation, the following criteria should be considered. First, it must be very simple and easy to use. Second, it must be gen-

[2] Readers are referred to Figure 12 in [16]

eral enough to capture most of the common cases that can benefit from tile percolation. Third, it should not bring much complexity to code generation and should also not cause too much runtime overhead. According to these criteria, the tile percolation programming API is designed as OpenMP pragma directives, which are shown in Figure 4(a).

```
#pragma omp percolate [tile ...]
#pragma omp tile ro (A$[j_{\dim(A)}, a_{\dim(A)}, L_{\dim(A)}]..[j_2, a_2, L_2][j_1, a_1, L_1], ...$)
                 wo (B$[k_{\dim(B)}, b_{\dim(B)}, M_{\dim(B)}]..[k_2, b_2, M_2][k_1, b_1, M_1], ...$)
                 rw (C$[l_{\dim(C)}, c_{\dim(C)}, N_{\dim(C)}]..[l_2, c_2, N_2][l_1, c_1, N_1], ...$)
```

(a) The definition of the tile percolation API

`percolate:`	Directive name. It specifies a percolation region
`tile:`	Directive name. It specifies a tile region and the tile descriptors
`ro:`	Clause name. It specifies the tiles that are read-only in the current percolation region.
`wo:`	Clause name. It specifies the tiles that are write-only in the current percolation region.
`rw:`	Clause name. It specifies the tiles that are read and written in the current percolation region.
`A,B,C:`	Name of the host multi-dimensional data array
j_i,k_i,l_i:	The index variable of the for loop that defines the i_{th} dimension of the tile
a_i,b_i,c_i:	Blocking size of the i_{th} dimension of the host multi-dimensional array (i.e. A, B, and C).
L_i,M_i,N_i:	Size of the i_{th} dimension of the host multi-dimensional array
$\dim(..)$:	The dimension of the multi-dimensional array

(b) The explanation of the tile percolation API

Fig. 4. The OpenMP API for tile percolation (C/C++)

The tile percolation API has two new pragma directives: the `percolate` directive and the `tile` directive. The percolate directive specifies a *percolation region*, which is a block of code. At the beginning of the percolation region, on-chip storage will be reserved for all data tiles that will be percolated into the on-chip memory. Then, all or some of the data tiles accessed in the percolation region will be moved into the on-chip memory and the corresponding computations will be performed there. At the end of the percolation region, data tiles that contain the results of the computations are written back to the off-chip memory (if necessary) and the reserved on-chip memory are freed.

The tile directive, on the other hand, provides the detailed information (type, shape, size, etc.) of the data tiles that will be percolated into the on-chip memory. It is always contained in a percolation region. The tile directive specifies a *tile region*, in which there is a set of `for` loops delimiting the bounds of the data tiles. In the tile directive, following the key word `tile` is a list of *tile descriptors*. The tile descriptors are divided into three groups by the key words **`ro`**, **`wo`**, and **`rw`**, which are the *clause* names that identify *read-only*, *write-only*, and *read-write* data tiles. At the beginning of the percolation region, data tiles specified in the **`rw`** clause will be copied from the off-chip memory into the on-chip memory (after the on-chip memory allocation). At the end of the percolation region, data tiles specified in the **`rw`** and **`wo`** clauses are copied back to their home locations in the off-chip memory. For data tiles specified in the **`ro`** clause, they will be copied into the on-chip memory at the place where the **`ro`** clause is specified. They will not be copied back to the off-chip memory at the end of the percolation region. The associated code in the percolation region are adjusted to access the on-chip data tiles in the computations.

The format of the tile descriptor is similar to the declaration of a multi-dimensional array variable, except that each of the tile descriptor's *dimension specifier* is a 3-tuple, not a singleton. The tile descriptor tells the compiler how the data tile is carved out from the multi-dimensional data array that hosts it. To make the paper easy to follow, we call the multi-dimensional data array that hosts the current data tile as its *host array*. The tile descriptor contains the complete information of the host array. Therefore, the number of dimension specifiers in the tile descriptor is the same as the dimension of the host array. It is not necessarily the same as the dimension of the data tile.

For a dimension specifier $[j_i, a_i, L_i]$ (see Figure 4(a)), ”L_i” is the size of the i_{th} dimension of the host array (not the data tile). ”a_i” is the blocking/tiling size of the i_{th} dimension of the host array. This parameter is used to carve out the data tile from its host array. Normally, if the dimension of the data tile is the same as the dimension of the host array, ”j_i” is the index variable of a **`for`** loop in the tile region that traverse the i_{th} dimension of the data tile. If the dimension of the data tile is smaller than its host array, the element j_i in some dimension specifiers becomes trivial. Currently, we force the programmers to put a ”$*$” there as a place holder to let the compiler know that the current dimension of the host array has been squashed away in the dimension space of the data tile. An intuitive example of this case is the expression `A[0][i][j]` guarded by loop `i` and loop `j`. It actually represents a 2-D plane, although the expression has 3 dimension specifiers.

The tile descriptor functions like a template and the associated `for` loops instantiate this template. To make the code generation easy, currently, a writable tile descriptor (specified in the **`rw`** or the **`wo`** clause) can only has one instantiation. The read-only tiles (specified in the **`ro`** clause) can have multiple instantiations. Example is given in Figure 5. To put it in a simple way, roughly, the percolate directive and the tile directive tell the compiler *where* the data tiles will be percolated and the tile descriptors tell the compiler *how* the data tiles are percolated. The code example that shows the usage of the tile percolation API is in Figure 5. The detailed explanation will be presented in the next sub-section.

3.2 Code Generation

The code in Figure 5 shows how to use the tile percolation API. The pragma at line 1 is the canonical `parallel for` directive that specifies how the computation iterations are distributed among the parallel threads. The pragma at line 5 is a percolate directive and line 8 is a tile directive. The percolate directive specifies a percolation region, from line 6 to 16. The tile directive specifies a tile region, from line 10 to 15, in which there are there data tiles, represented by ”`A[i,b,n][k,b,n]`”, ”`B[k,b,n][j,b,n]`”, and ”`C[i,b,n][j,b,n]`”. The first two tiles are read-only and the last one is both readable and writable in the current percolation region. They direct the compiler to generate the correct data percolation and computation code.

The tile descriptor ”`C[i,b,n][j,b,n]`” specifies a data tile contained in the host array `C`, a 2D $n \times n$ matrix. In this tile descriptor, ”`C`” provides the name of the host array, which also tells the compiler the type of the data element of the tile. ”`n`” in

```
 1 #pragma omp parallel for collapse(2)
 2 for (ii=0; ii<n; ii+=b)
 3   for (jj=0; jj<n; jj+=b)
 4   {
 5     #pragma omp percolate
 6     {
 7       for (kk=0; kk<n; kk+=b)
 8       #pragma omp tile ro (A[i,b,n][k,b,n],B[k,b,n][j,b,n]) \
 9                        rw (C[i,b,n][j,b,n])
10       {
11         for (i=ii; i<min(ii+b,n); i++)
12           for (j=jj; j<min(jj+b,n); j++)
13             for (k=kk; k<min(kk+b,n); k++)
14               C[i][j]+=A[i][k]*B[k][j];
15       }
16     }
17   }
```

Fig. 5. Pseudo Code of the Tile Percolation Example

the dimension specifier tells the size of the each dimension of the host array. ”b” reveals how the matrix is tiled. ”i” and ”j” are two index variables that inform the compiler that the for loops at line 11 and 12 are used to construct the data tile. Since the lower and upper bounds of ”i” and ”j” are fixed in the current percolation region, there is only one instantiation for this tile template (i.e. descriptor). The clause name ”rw” indicates that this data tile will be read and written in the current percolation region. So, it will be copied into the on-chip memory at line 6, where the percolation region starts. It will also be copied out to off-chip memory at line 16, where the percolation region ends.

Similarly, data tile ”A[i,b,n][k,b,n]” and ”B[k,b,n][j,b,n]” are contained in host array ”A” and ”B”, which are also 2D $n \times n$ matrix. Since both of them are read-only data tiles, they are copied into the on-chip memory at line 8, where they are specified in the **ro** clause. They do not need to be copied back to the off-chip memory at the end of the percolation region. Because the lower and upper bounds of ”k” are changing (line 7), as we may notice, there are multiple instantiations for these two tile descriptors. All instantiations of the same data tile will reuse the same memory block allocated to it. The example is shown in Figure 6.

Figure 6 presents the code generated for the tile percolation program in Figure 5. First, it allocates on-chip memory for all three data tiles (line 5 to 7). This is achieved by calling the runtime routine _sram_malloc, which is inserted in by the compiler. The size of the data tile is calculated by multiplying each of its dimension size, which is obtained from its blocking size. This guarantees that the memory block allocated is big enough to hold the corresponding data tile. If the memory allocations succeed, the read-write data tiles will be copied into the on-chip memory by calling the runtime library routine _copy2Don (line 16). Otherwise, no data movement happens and the program execution jumps to the original code (line 12), where computations are performed on off-chip data tiles (line 36).

The other two read-only data tiles are percolated into the on-chip memory between the for loops at line 18 and 25. This location corresponds to the place in the original code where they are specified in the **ro** clause. The for loops between line 25 and 28 perform matrix multiplication on ”_AA[][]”, ”_BB[][]”, and ”_CC[][]”, which

```
 1 {
 2   /* Enter the percolation region.
 3    * Allocate on-chip memory for all data tiles
 4    */
 5   _CC=(float *)_sram_malloc(sizeof(float)*b*b);
 6   _AA=(float *)_sram_malloc(sizeof(float)*b*b);
 7   _BB=(float *)_sram_malloc(sizeof(float)*b*b);
 8
 9   if (_CC==NULL || _AA==NULL || _BB==NULL)
10   {
11     _sram_free(_AA); _sram_free(_BB); _sram_free(_CC);
12     goto orig;
13   }
14
15   /* Copy "rw" data tiles from off-chip memory to on-chip memory */
16   _copy2Don(sizeof(float),_CC,&C,n,n,ii,jj,min(b,n-ii),min(b,n-jj));
17
18   for (kk=0; kk<n; kk+=b)
19   {
20     /* Copy "ro" data tiles from off-chip memory to on-chip memory */
21     _copy2Don(sizeof(float),_AA,&A,n,n,ii,kk,min(b,n-ii),min(b,n-kk));
22     _copy2Don(sizeof(float),_BB,&B,n,n,kk,jj,min(b,n-kk),min(b,n-jj));
23
24     /* on-chip calculation */
25     for (i=0; i<min(b,n-ii); i++)
26       for (j=0; j<min(b,n-jj); j++)
27         for (k=0; k<min(b,n-kk); k++)
28           _CC[i][j]+=_AA[i][k]*_BB[k][j];
29   }
30
31   /* copy out the results back to off-chip memory */
32   _copy2Doff(sizeof(float),_CC,&C,n,n,ii,jj,min(b,n-ii),min(b,n-jj));
33   _sram_free(_AA); _sram_free(_BB); _sram_free(_CC);
34   goto out;
35
36 orig:
37      /* Original code with out percolation */
38         ...
39 out:
40 }
```

Fig. 6. Code generation example for tile percolation (Pseudo Code)

are all resided in the on-chip memory. After one kernel computation (line 25 to 28) is finished, the new instantiation of ”`_AA[][]`” (and also ”`_BB[][]`”) is copied from the off-chip memory into the on-chip memory and is stored in the same memory block. Then it begins the next iteration. Before exiting the percolation region, the **`rw`** data tile ”`_CC[][]`” is copied back to its home location in the off-chip memory (line 32). Meanwhile, the on-chip memory storage allocated to all the percolated data tiles are freed.

To generate the code like in Figure 6, the compiler needs to handle three tasks: **(1)** generate code for managing on-chip memory; **(2)** generate code for managing memory copy; **(3)** adjust the computation code to access on-chip data tiles. We leave the discussion of the first two items to the next sub-section, because they are mostly related to the runtime. Here, we focus on the third problem.

Adjusting the computation code to access on-chip data tiles includes two sub-tasks: **(i)** calibrate the lower and upper bounds for each `for` loop that is involved in traversing the elements of the data tile; **(ii)** update the tile access expressions accordingly. These tasks are easy because the data tile is copied as one 2D array from its home location (in which, the elements are physically scattered in memory) into a piece of physically contiguous memory block (in which, the elements are consecutive). We only need to know the base address of the memory block and the size of each dimension of the data tile. The value of the tile's dimension size can be easily obtained from its tile descriptor. The base address of the memory block that assigned to the current data tile can be obtained from the corresponding runtime function call (`_sram_malloc`). With this information, it is easy for the compiler to generate the correct code. Most of time, we just perform a kind of simple 1-to-1 replacement. For example, the new lower bound of a `for` loop is always set to zero and the new upper bound is calculated by subtracting the old lower bound from the old upper bound.

3.3 Runtime Routines and Runtime Support

As we have mentioned in the last section, the tile percolation runtime needs to handle the on-chip memory allocation and the memory copy for the percolated data tiles. We provide a set of routines (with clear interface) in the runtime library for the compiler. The compiler, accordingly, would insert the required runtime function calls in the program during code generation.

The runtime routines `_sram_malloc` and `_sram_free` are responsible for allocating on-chip memory for the percolated data tiles. To allocate the correct memory storage for the tile, we need to know three values: (i) the number of dimensions of the tile; (ii) the size of each dimension; and (iii) the type of each data element. The number of dimensions of the tile is the number of non-trivial dimension specifiers in its tile descriptor. The dimension size is always set to the blocking size. This guarantees that the allocated memory block is big enough to hold any instantiation of the tile descriptor. The type of the data element is obtained from the name of the tile descriptor.

For each percolation region, the ”all-or-none” policy is adopted in memory allocation. The program either continues execution after *all* of its memory allocation requests were satisfied, or, if *any* of its memory allocation request failed, it jumps to the original code to perform the computations on the off-chip data tiles. Because the compiler guarantees that all memory allocations occur at the beginning of the percolation region and all memory frees occur at the end of the percolation region, the memory allocation failure would not cause dead lock among the concurrent OpenMP threads. This greatly simplifies design of the runtime support and also simplifies code generation in compiler.

For the memory copy task, we provide the set of runtime routines presented in Figure 7. Currently, we support tile percolation for 1D-, 2D-, and 3D-array. They can cover most of the practical cases. Each kind of multi-dimensional array has its own memory-copy routines (see Figure 7(a)). The routines with the suffix ”*on*” are used to copy data tiles from off-chip memory to on-chip memory, while the routines with the suffix ”*off*” are used to copy data tiles from on-chip memory to off-chip memory. The parameters that are required in the address calculation in these memory-copy routines are supplied in the argument list. We use the ”long argument list” instead of the ”packed argument structure” because we try to avoid inserting unnecessary dynamic memory allocation function calls in code generation. We feel that generating dynamic memory allocation code is tricky and error-prone.

```
_copy1Don(sz,_on,_off,D1,x,b1)
_copy1Doff(sz,_on,_off,D1,x,b1)
_copy2Don(sz,_on,_off,D1,D2,x,y,b1,b2)
_copy2Doff(sz,_on,_off,D1,D2,x,y,b1,b2)
_copy3Don(sz,_on,_off,D1,D2,D3,x,y,z,b1,b2,b3)
_copy3Doff(sz,_on,_off,D1,D2,D3,x,y,z,b1,b2,b3)
 ...
```

(a) The runtime routines for memory copy

`_copy[1D|2D|3D]on`:	Runtime routines that copy the off-chip data tile into the on-chip memory;
`_copy[1D|2D|3D]off`:	Runtime routines that copy the on-chip data tile back to the off-chip memory;
`sz`:	Size of the element of the data tile;
`_on`:	The address of the on-chip memory block used to hold the percolated data tile;
`_off`:	The address of the home location of the percolated data tile in the off-chip memory;
`D1,D2,D3`:	The size of each dimension of the percolated data tile, from the lowest dimension to the highest dimension;
`x,y,z`:	Position of the percolated data tile in the host array. It is represented by the coordinate of its first element in the host array, from the lowest dimension to the highest dimension;
`b1,b2,b3`:	Blocking size of each dimension of the host array, from the lowest dimension to the highest dimension;

(b) The explanation of the runtime routines

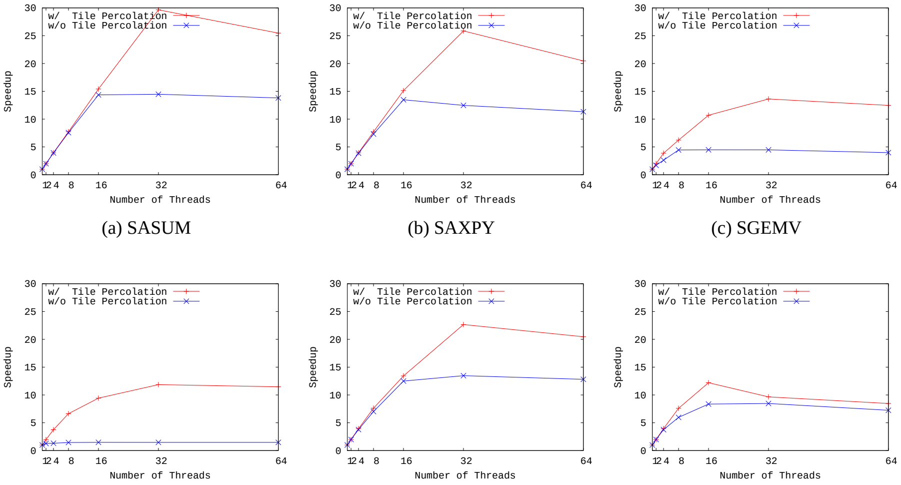
Fig. 7. The runtime routines for on-chip and off-chip memory copy

According to our design, there are some assumptions on the on-chip and the off-chip data tiles. For the on-chip data tile, it must reside in a contiguous memory block. For the off-chip data tile, it must be a sub-block of a multi-dimensional array and the multi-dimensional array must also reside in a contiguous memory block. Because the percolated data tile is only a sub-block in its host array, its memory layout is not contiguous. Physically, it consists of many data strips (or rows) that are separated by an equal distance. Hence, the parameters provided in the argument list should be able to be used to calculate the start address and the size of each data strip in the tile. With the above assumptions, it is easy to interpret the argument list of the memory-copy routines. For example, the routine `_copy2Don` copies a 2D data tile from off-chip memory to on-chip memory. The argument ”`_off`” gives the start address of its host array, (i.e. the address of the first element). ”`D1`” and ”`D2`” tell the dimension size of the array. ”`x`” and ”`y`” specify the coordinates of the data tile in its host array. ”`b1`” and ”`b2`” reveal the blocking size of each dimension of the host array. ”`b1`” and ”`b2`” are the default size (of each dimension) of the percolated data tile. To handle the corner cases, the routine will calculate the effective size at runtime. The size of the data tile element is shown in ”`sz`”. With the above information, it is easy for the runtime routine to calculate the address and size of each data strip and copy it around with the optimized library code. All these arguments are provided in the tile percolation directives and can be easily extracted out by the compiler.

In essence, the arguments listed above characterize the position and the size of a data tile and its host array accurately. It doesn’t matter whether this data tile and its host array are real multi-dimensional array (in the language sense) or not. As long as all the array access expression are affine functions of the loop indices, they can be declared (physically) as an 1D array but accessed by the programmers (logically) as a multi-dimensional array. The compiler will take care of the convoluted indices calculation.

4 Experiments

We evaluated tile percolation with four scientific kernels (SAXPY, SASUM, SGEMV, and SGEMM) [17] and two NAS benchmarks (EP and MG). The tile percolation was implemented through source-to-source program transformation and was prototyped in the Omni compiler [18]. The experiments were conducted on the FAST simulator [2], an execution-driven and binary-compatible C64 simulator with accurate instruction timing. Figure 1(b) gives the detailed latency numbers of the load/store operations when accessing different memory segments. The preliminary experiment results are shown in Figure 8. Due to the space limit, we only present the performance speedup of each testcase.



(a) SASUM

(b) SAXPY

(c) SGEMV

(d) SGEMM

(e) EP

(f) MG

Fig. 8. Experiment Results: Comparison of Speedup

After applying tile percolation, the speedup of all testcases get significant improvement. The greatest improvement happens on SGEMM [3]. This testcase has $O(n^3)$ floating-point operations but only access $O(n^2)$ data. However, without reusing the data that have been brought into on-chip memory by the previous computations, the program has very poor scalability. Because it would have $O(n^3)$ number of memory accesses going into the off-chip memory. This would quickly exhaust the off-chip bandwidth. Without using on-chip memory, its diminishing return is 2-thread. After applying the tile percolation optimization, the number of memory accesses has been reduced to $O(n^2(1 + 2n/b))$. Its speedup increased from less than 4 to around 12. For other testcases, their floating-point computations are $O(n^2)$ (SGEMV) or $O(n)$ (EP). So their speedup enhancement is not as big as SGEMM.

[3] We use 256×256 matrix, the data tile is 16×16

An interesting finding is that, without applying tile percolation, most testcases' speedup diminishing return point is at 16-thread. They are SASUM, SAXPY, EP, and MG. The speedup diminishing return point of SGEMV is 8-thread, while for SGEMM, it is 2-thread. For SASUM, its memory accesses and floating-point operations are the same. This reveals that, without on-chip data reuse, the off-chip bandwidth would be saturated when there are more than running 16 threads.

5 Summary and Conclusions

Writing a parallel program for multicore processor is already a very difficult task. It is even more difficult if the multicore processor has software managed memory hierarchy, like the IBM Cyclops-64 processor. On this kind of processor, the programmers not only need to take care of program parallelization, but also need to tackle data movement. Although many efforts have been made to develop automatic data movement code generation, it only proves its efficiency on a limited class of problems.

In this paper, we have proposed a semi-automatic approach to data movement code generation. This novel approach is termed as *tile percolation*. It provides the programmers with a set of OpenMP-like directives. The programmers can annotate the their programs with these directives to tell the compiler *where* and *how* data movement should be performed. Accordingly, the compiler will generate the optimized data movement code and the correct computation code based on the information provided in the tile percolation directives. That way, the programmers can save themselves from writing tedious and error-prone data movement code.

Tile percolation is a kind of OpenMP *tile aware parallelization* technique [7] developed for the IBM Cyclops-64 multicore processor. As far as the authors are aware, this is the first paper that try to develop pragma directives for data movement code generation in OpenMP. The tile percolation directives are orthogonal to the canonical OpenMP parallelization directives. The paper shows that the tile percolation directives can be used together with the traditional OpenMP parallelization directives. Meanwhile, they can also be used independently in the parallel programs written with Pthread library. Experiments conducted on the Cyclops-64 processor show that tile percolation can enhance the utilization of the Cyclops-64 on-chip memory, which turns out to improve the performance and scalability of the programs. This implements an efficient *performance porting* for OpenMP programs developed for the traditional SMP system.

References

1. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 3.0 (May 2008) http://www.openmp.org/mp-documents/spec30.pdf.
2. 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)
3. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Towards a software infrastructure for cyclops-64 cellular architecture. In: HPCS 2006, Labroda, Canada (June 2005)

4. Zhang, Y., Jeong, T., Chen, F., Wu, H., Nitzsche, R., Gao, G.R.: A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture. In: IPDPS'06: Proceedings of the 20th International Parallel and Distributed Processing Symposium, 25-29 April 2006, Rhodes Island, Greece. (April 2006)
5. Hu, Z., del Cuvillo, J., Zhu, W., Gao, G.R.: Optimization of dense matrix multiplication on ibm cyclops-64: Challenges and experiences. In: Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, August 28 - September 1, 2006, Proceedings. (2006) 134–144
6. del Cuvillo, J., Zhu, W., Gao, G.: Landing openmp on cyclops-64: an efficient mapping of openmp to a many-core system-on-a-chip. In: CF '06: Proceedings of the 3rd conference on Computing frontiers, New York, NY, USA, ACM (2006) 41–50
7. Gan, G., Wang, X., Manzano, J., Gao, G.R.: Tile reduction: the first step towards openmp tile aware parallelization. In: Lecture Notes in Computer Science: OpenMP in a New Era of Parallelism, IWOMP'09, International Workshop on OpenMP, Springer Berlin / Heidelberg (2009)
8. Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Croz, J.D., Hammerling, S., Demmel, J., Bischof, C., Sorensen, D.: Lapack: a portable linear algebra library for high-performance computers. In: Supercomputing '90: Proceedings of the 1990 conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society Press (1990) 2–11
9. Anderson, E., Dongarra, J.J.: Evaluating block algorithm variants in LAPACK. Technical Report 19, LAPACK Working Note (April 1990)
10. Ltaief, H., Kurzak, J., Dongarra, J.: Parallel block hessenberg reduction using algorithms-by-tiles for multicore architectures revisited. Technical Report 208, LAPACK Working Note (August 2008)
11. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.J.: Parallel tiled QR factorization for multicore architectures. Technical Report 190, LAPACK Working Note (July 2007)
12. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report 191, LAPACK Working Note (September 2007)
13. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. Society for Industrial and Applied Mathematics, Philadelphia (1998)
14. Chen, T., Zhang, T., Sura, Z., Tallada, M.G.: Prefetching irregular references for software cache on cell. In: CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization, New York, NY, USA, ACM (2008) 155–164
15. Chen, T., Lin, H., Zhang, T.: Orchestrating data transfer for the cell/b.e. processor. In: ICS '08: Proceedings of the 22nd annual international conference on Supercomputing, New York, NY, USA, ACM (2008) 289–298
16. Lee, J., Seo, S., Kim, C., Kim, J., Chun, P., Sura, Z., Kim, J., Han, S.: Comic: a coherent shared memory interface for cell be. In: PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, New York, NY, USA, ACM (2008) 303–314
17. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program gpus for general-purpose uses. In: ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, New York, NY, USA, ACM (2006) 325–335
18. 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