Cache efficient parallelizations for Uniform Dependence Computations

Yun Zou Sanjay Rajopadhye

May 1, 2014

Colorado State University Technical Report CS-14-101

Abstract

Decreasing the traffic from processors to main memory is beneficial in many aspects, such as reducing energy consumption and bandwidth demand, increasing device lifetime, etc. We address a class of programs that are stencil like computations, for which, tiling is a key transformation. The target platform is a shared memory multi- and many-core processor. We propose a cache-efficient parallelization strategy that uses two-level tiling with multiple passes and seeks to minimize the total number of off-chip memory accesses. We mathematically analyze the execution time and main memory accesses, and use it to guide the choices for all the tiling parameters. We also develop a tiled code generator to support this parallelization strategy, which we validate experimentally on a number of stencil benchmarks including those from the well known Polybench suite. On a 8-core Intel Xeon E7-4830 based machine and a 6-core Xeon E5-2620 v2 based machine, our results on some benchmarks show more than 40-fold reduction in off-chip memory accesses with less than 10% slow-down. With around 30% slow-down, we can achieve up to 60-fold reduction in off-chip memory on Intel Xeon E7-4830, and more than 70-fold reduction on Xeon E5-2620 v2. We also describe additional optimizations that we are currently implementing, to recover the execution time slow-down that is introduced by our parallelization strategy.

1 Introduction

Off-chip memory access performs a critical role in a computer system's performance and energy efficiency [39, 23, 36]. A modern multi-core processor provides a shared on-chip memory that is shared by a number of cores. The cores can execute distinct applications or cooperate for the same application by using the shared memory for communication. The shared on-chip memory is usually the last level cache, and a cache miss of the last level cache leads to off-chip memory transfer. In this paper, we focus on reducing the cache misses of the last level shared on-chip memory.

Exploring the memory hierarchy efficiently is the key to reduce the memory traffic. Much research has been done to explore the efficiency of the memory hierarchy [36, 22]. Tiling [36, 38] is a critical program transformation and is used extensively to address many different aspects of performance, including off-chip memory accesses. One important class of programs to which tiling can be applied is called *stencil computations*. One of the thirteen Berkeley dwarfs/motifs (see `http://view.eecs.berkeley.edu`), is "structured mesh computations," which are essentially stencils. The importance of stencils has been noted by a number of researchers, indicated by the recent surge of research projects and publications on this topic, ranging from optimization methods for implementing such computations on a range of target architectures, to Domain Specific Languages (DSLs) and compilation systems for stencils. A number of workshops and conferences focus specifically on stencil acceleration.

Stencil computations occur frequently in a wide variety of scientific and engineering applications, such as environment modeling applications that involve

Partial Differential Equations (PDE) solvers [30], computation electromagnetic applications using Finite Difference Time Domain (FDTD) methods [33], and computations based on neighboring pixels and multimedia/image-processing application [11]. Naive implementations of these applications turn out to be memory-bound. Therefore, much work has been done to explore the data locality for stencil computations [16, 26, 36, 37]. One successful technique is called *time skewing* [36, 8, 37]. With time skewing, the whole computation space of a stencil can be tiled and executed in a wavefront fashion. Although this technique can greatly improve the data locality for stencil programs, there is still a significant amount of off-chip data transfer for stencil programs with large problem size, which can be further reduced.

In this paper, we propose a cache-efficient parallelization strategy for stencil programs. We assume, like most of the work in the literature, that *dense* stencil programs are inherently *compute bound*—in particular, any global reductions, such as those in convergence/termination tests can be safely removed. Our parallelization strategy is based on multi-level tiling technique associated with a *multi-pass* execution. The multi-pass execution strategy [24] was introduced to address the problem of restricted resources in the context of systolic array, and later it was adapted to address the resource constraints on different platforms [29]. Here, we adapt the multi-pass execution strategy to address the last level cache misses on general multi-core processors. Although we claim that our strategy is designed for stencil computations, it works for a much larger set of programs. In general, our strategy works on computations described as Systems of Uniform Recurrence Equations [17]. These are a subset of programs that can be analyzed with polyhedral techniques, and on which tiling transformations can be applied. The platform we currently target is a single multi-core processor.

We declare our contributions as follows:

- We introduce a cache-efficient parallelization strategy for stencil like computations. Our parallelization strategy focuses on reducing the total number of off-chip memory accesses.

- We mathematically analyze the execution time and off-chip memory access for our parallelization strategy. We use the analysis to show the trade-offs between the off-chip memory transfer and execution time and guide the choice of optimal tile size selection.

- We develop an automatic code generator based on polyhedral techniques to support our parallelization strategy.

- We evaluate our parallelization strategy on a set of well known benchmarks, in comparison with the standard tiled parallelization. On two of them we get a 30× reduction in the number of last level cache misses for around 30% slowdown in execution time on Xeon E7-4830. On three others, the savings is over 40× and the slowdown is only about 10%. We get even better trade-offs on Xeon E5-2620 v2. For the Smith-Waterman dynamic programming algorithm on very large sequences, the slowdown is again only 10% but the savings is over 7000$\times$ on Xeon E7-4830 and 3000 on Xeon E5-2620 v2, three orders of magnitude!

This paper is organized as follows: Section 2 briefly describes some background about off-chip memory access and polyhedral model. Section 3 introduces our cache-efficient parallelization strategy and our tiled code generator. Section 4 mathematically describes the trade-offs between execution time and off-chip memory transfer. Section 5 shows the results of our experiments on a set of benchmarks. Section 6 discusses some related existing work. Finally, in Section 7, we present our conclusion and future work.

2 Background

We now describe some background and terminology needed for the rest of paper.

2.1 Cache misses

An off-chip memory access is caused by a failed request of read or write to the cache, which is called a *cache miss*. The cache misses are classified into three categories [14]: compulsory miss, capacity miss, and conflict miss.

Compulsory miss. A compulsory miss occurs on the first access to a memory location, since at the very beginning of execution, no data is in the cache. Such misses are also called *cold misses*.

Capacity miss. A capacity miss occur when the total amount of memory used by a program is larger than the cache capacity. In this situation, the cache is not large enough to hold all the data that is needed, Therefore, some data has to be evicted from the cache, and a request to this data will cause a miss.

Conflict miss. A conflict miss occurs for an associative cache or directly mapped cache (a directly mapped cache is a special case: 1-way associative cache). For an N-way associative cache, the whole cache is divided into sets, each set can hold N distinct cache lines, and each cache line is mapped to one set. Conflict miss occurs when the program frequently accesses more than N distinct cache lines that are mapped into the same set. In this paper, we do not count conflict misses.

2.2 Stencil computation

A stencil computation [16, 26] is a computation that repeatedly updates each point in a d-dimensional grid over T time steps. Therefore, a stencil computation of d-dimensional data grid has $(d+1)$-dimensional iteration space. At a time step t, the computation for each point is defined as a function of its neighboring points at previous time steps or possibly the current time step.

A stencil computation is called an n-point stencil computation if each point is defined as a function of n neighboring points. The *order* of a stencil computation is defined as the distance of the furthest grid point in the neighboring points.

For example, Jacobi 1D is a 3-point first-order stencil computation. The computation is $u_{t,i} = a(u_{t-1,i-1} + u_{t-1,i} + u_{t-1,i+1})$, to compute each point at a time step requires three neighboring points from the previous time step.

Although the examples and benchmarks described in this paper are all stencil computations, our strategy is applicable to a more general class of programs – programs with uniform dependences and to which tiling transformations can be applied.

2.3 Polyhedral model

The polyhedral model is a mathematical formalism for analyzing, parallelizing and transforming an important class of compute- and data-intensive programs. Although it is often used as the foundation for automatic parallelization of an important class of loop programs, its scope is wider that that and includes equational programming (with recurrences) and analysis of mathematical properties of programs. The polyhedral model is also very useful in automatic parallelization [6, 4, 27].

Domain: Each computation statement in a program is surrounded by loops with affine bounds. The domain of a statement describes the iteration space in which the statement is defined and is represented by a set of linear inequalities. For example, the domain of the Jacobi 1D computation is $\{i, t \mid 0 \leq i \leq N, 0 \leq t \leq T\}$, where i, t are the loop index names, N is the upper bound of the i dimension and T is the upper bound of the t dimension.

Dependence. Two iterations S_i and S_j are said to be dependent if they access the same memory location and one of the accesses is a write. For example, in Jacobi 1D, iteration (t, i) depends on iteration $(t-1, i)$, $(t-1, i+1)$ and $(t-1, i-1)$.

3 Cache-Efficient parallelization strategy

In this section, we describe our cache-efficient parallelization strategy that is called multi-pass strategy. We use Smith-Waterman as an example to illustrate our parallelization strategy and its memory behavior. In general, our strategy works for general polyhedral programs with continuous dimensions only involves uniform dependences and fully permutable.

3.1 Smith-Waterman

Smith-Waterman [1] is a well-known algorithm for biological sequence comparison that is used to find the optimal local sequence. The algorithm does the

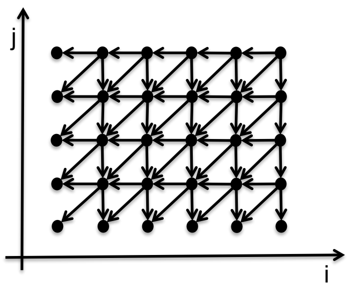

Figure 1: Dependence graph for Simith-Waterman. (The source of an arrow is a consumer and the destination of the arrow is a producer)

following computation:

$$H_{i,0} = 0, \quad 0 \leq i \leq m$$
$$H_{j,0} = 0, \quad 0 \leq j \leq n$$
$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + w(a_i, b_j) \\ H_{i-1,j} + w(a_i, -) \\ H_{i,j-1} + w(-, b_j) \end{cases} \quad 1 \leq i \leq m, 1 \leq j \leq n$$

Where a and b are two strings, m is the length of a and n is the length of b. The function w computes a score. Figure 1 shows the dependence relationship between computations. The computation of each point (i, j) depends on the values from three neighboring points: $(i, j-1)$, $(i-1, j)$, and $(i-1, j-1)$.

3.2 Standard tiling and parallelization

For Smith-Waterman, standard tiling and wavefront parallelization strategy can be applied to optimize the performance, and many work have been published for optimizing Smith-Waterman on different parallel platforms [21, 1, 7]. Figure 2 illustrates the standard tiling and wavefront parallelization for Smith-Waterman. Given a Smith-Waterman problem with problem size $M \times N$ and tile size $x \times y$. The number of tiles in the graph is $m \times n$, where $m = \lceil \frac{M}{x} \rceil$ and $n = \lceil \frac{N}{y} \rceil$. When $M \gg x$ and $N \gg y$, $\lceil \frac{M}{x} \rceil \approx \frac{M}{x}$ and $\lceil \frac{N}{y} \rceil \approx \frac{N}{y}$. In this paper, we consider stencil computations with large problem sizes, therefore, we can ignore the ceiling in our following analysis. Tiles are executed in a wavefront fashion, one wavefront is started after the previous wavefront is done and all the tiles inside one wavefront can be started in parallel.

Now, let's quantify the off-chip memory accesses for the Smith-Waterman with standard tiling and wavefront parallelization. To perform the computation in each tile, it requires x values from the tile above it and y values from the tile

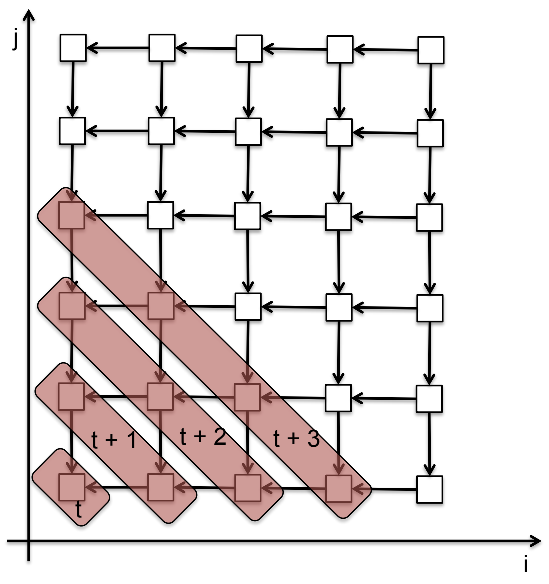

Figure 2: Tiling and wavefront parallelization for Smith-Waterman. Each square represents a tile with width x and height y, and each red diagonal strip represents one wavefront pass. Each wavefront is started after the previous wavefront is done, and all the tiles in one wavefront can be computed simultaneously.

at left. And it produces $x + y$ values for the tiles that are executed in the next wavefront. Assume that the problem size is large enough that the size of the values produced at one wavefront is much larger than the capacity of last level cache. Then every read of the values produced from the previous wavefront is a compulsory miss, and it will cause an off-chip memory access. Therefore, the total volume of off-chip memory transfer is $V_1 = \frac{M}{x} \times \frac{N}{y} \times (x + y) = MN\frac{(x+y)}{xy}$.

Another strategy that is designed to use the memory hierarchy efficiently is called *multi-level tiling* [12, 29]. The multi-level tiling strategy hierarchically divided the computation space with the idea that the base tile can fit into a deeper cache level. The wavefront parallelization strategy is used on the outermost tile level or all the tile levels. However, under the worst case, every read of the value produced from the previous wavefront is still going to be a miss for the outermost level tiles. Next, we will introduce our cache-efficient multi-pass parallelization strategy.

3.3 Cache-efficient parallelization

As we mentioned above, the values produced at one wavefront is going to be fed into the next wavefront directly. And when the number of the values produced at one wavefront is large enough to exceed the last level cache, each read of those values at next wavefront is going to be a miss. Therefore, reusing the values produced from the previous wavefront is the key to reduce the volume of off-chip data transfer.

To address the reuse between adjacent wavefronts, we propose an multi-pass parallelization strategy based on multi-level tiling. As illustrated in Figure 3, our multi-pass parallelization strategy first tiles the computation space into passes, and then further tiles the computation in each pass into smaller tiles. The passes are executed sequentially, and inside each pass, standard wavefront parallelization is applied.

If we choose the pass height carefully that the number of values produced by one wavefront in one pass can fit into the last level cache, then the read of those values for the next wavefront can still remains on chip, but cold misses still happens at the boundary of each pass. Let's assume the problem size is $M \times N$, the tile size for the tiles inside each pass is $x \times y$, and the height for each pass is H. Assume H is small enough that the value produced at one wavefront in one pass can fit into the last level cache. Then, the volume of off-chip data transfer can be estimated as the total volume of the boundary of all the passes, which is $V_2 = (M + H) \times \frac{N}{H}$.

Now, let's compare the volume of off-chip memory transfer of our multi-pass parallelization strategy with the standard wavefront parallelization strategy. $\frac{V_1}{V_2} = \frac{MH}{(M+H)} \times \frac{x+y}{xy}$. When $x = y$ and $M = H$, this formula yields the maximum value $\frac{H}{x}$, and H is usually greater than x. If H is much larger than x, our strategy can potentially reduce the off-chip memory access significantly.

This multi-pass strategy can be generalized to higher dimension. For problems with d dimensions ($d \geq 2$), we first tile k dimensions ($k < d$) to get the

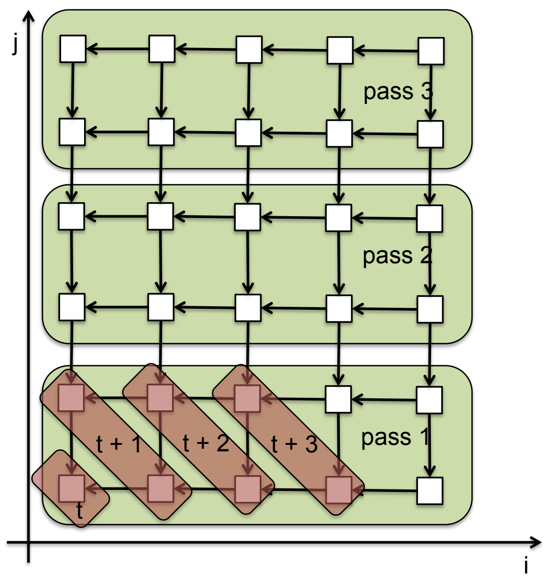

Figure 3: Multi-pass parallelization for Smith-Waterman.

passes, and then perform an inner level tiling to all the dimensions. Figure 4 shows an example with three dimensional computation space ($d = 3$). If dimension i and dimension j are tiled first, the whole computation space is divided into a set of tubes with the inner most level not tiled (shown in the right of Figure 4), and each tube is executed one after another sequentially. For each tube, one more level of tiling that tiles all the three dimensions is applied, all the tiles inside a tube is executed in a wavefront fashion.

3.4 Code generation

It is known that tiled code is hard to write, therefore, much work have been done to develop code generator that generates tiled code automatically, such as HiTLOG [35, 18], Pluto [6], PrimeTile [13] and etc. Most existing code generators support multi-level tiling [18, 6, 13] that tile all the dimensions, and some of them support [18, 6] wavefront parallelizations for outer level of tiles and Pluto also supports wavefront parallelization for all tile levels [6]. However, our parallelization strategy does not require all the dimensions to be tiled, although we can set the tile size to be the problem size, possible loop overhead can be introduced. Moreover, we require the support of wavefront parallelization at inner level, but with sequential execution at the outer level.

We developed a Parameterized Hierarchical Tiled Code generator (PHTile) that generates parameterized hierarchical tiled code for shared memory architecture. Our code generator takes a polyhedral specification of a polyhedral

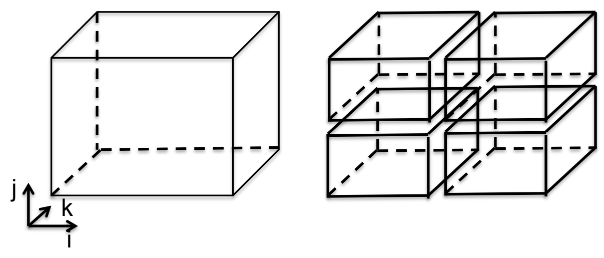

Figure 4: Multi-pass strategy for 3D program, the outer two dimension is tiled for passes.

```
//outer non-tiled loops
for(Li = lbi; Li < ubi; Li++){
  . . .        //loops
  //tile band: the set of loops
  //that are going to be tiled
  for(Lj = lbj; Lj < ubj; Lj++){
          . . . //loops
        //inner non-tiled loops
    for(Lk = lbk; Lk < ubk; Lk++){
          . . . //loops
    }
  }
}
```

Figure 5: The structure of a tile band. The non-tiled loops are sets of loops that are not tiled. The tile band specifies the loops are going to be tiled.

program and produces C + OpenMP code. It tiles sets of continuous dimensions called *tile band* and supports wavefront parallelization at any level of tiling.

Figure 5 gives an example of a tile band. A program can have multiple tile bands and each tile band can be tiled for k levels (0 is the outermost level), and we require the inner level tiling tiles a subset of the dimensions of the outer level tiling. The dimensions to be tiled for different bands have to be disjoint.

The tiling process goes from band to band, and for each band, tiling is applied from the outermost to the innermost level. At each level of each band, the tiling algorithm described in Kim's work [18] is used to generate tiled code or tiled code with wavefront parallelization. A set of tiled-loops and a set of point-loops are generated separately for the tiling loops. The tiled-loops enumerates the tiled origins along the tiled dimensions and the point-loops enumerates all the points inside the tile along the tiled dimensions. The point-loops generated at one level are used to generate loops for the next level.

To visit the tiles with wavefront parallelization strategy, a loop is generated to visit all the wavefront time step. In each time step, loops are generated to visit all the possible tiles at current time step *time*. For a given d-dimensional tile with tile origin $(t_1, t_2 \ldots t_d)$, the time step to execute the tile is $time = \sum_{k=1}^{d} \frac{t_k}{s_k}$ where s_k is the tile size for the kth dimension. The lower bound and upper bound for the time steps are computed according to the bounds of the tiled loops, the computation is shown below

$$time_{start} = \sum_{k=1}^{d} \left(\frac{lb_k}{s_k} \right)$$

$$time_{end} = \sum_{k-1}^{d} \left(\frac{ub_k}{s_k} \right)$$

where lb_k is the lower bound of the kth loop, and ub_k is the upper bound of the kth loop. Those are all described in Kim's work. For tiling at inner levels, the point loops generated from the outer level is used as the input to the tiling algorithm.

We can also deduce that the total number of wavefront time steps n_w at a given level is

$$\begin{aligned} n_w &= time_{end} - time_{start} \\ &= \sum_{k=1}^{d} \left(\frac{ub_k - lb_k}{s_k} \right) \\ &= \sum_{k=1}^{d} \left(\frac{N_k}{s_k} \right) \end{aligned}$$

where N_k is the loop size of dimension k.

In our code generation frame work, the compilation is based on two main step.

- First, an internal data structure called *target mapping* specifies, for each statement in the program, all the aspects of the desired parallelization. This includes the schedule (i.e. time skewing), the memory allocation, and tiling—which dimensions to tile, how many levels of tiling, and whether wavefront parallelization is applied at a certain level.

- Next, a code generator in the form of a very sophisticated "pretty printer" based on CLooG [5] actually produces the target code.

The choice of the target mapping is a long standing research problem, since target architectures are continually evolving. Therefore, we require the users to input the target mapping currently.

4 Performance analysis

In this section, we mathematically analyze the execution time and off-chip memory accesses for the standard wavefront parallelization strategy and our mulch-pass strategy.

4.1 Execution time analysis

Some research has been done to model the execution time for tiled programs [19, 25, 2]. Our analysis here is based on the work of Andonov et al [2]. We first present the execution time analysis for the standard tiling and wavefront strategy, then we will extend it to support our multi-pass strategy.

The execution time for a given stencil program is the sum of time spent in each wavefront. Let n_w represent the total number of wavefronts, and $T(w_i)$ be the execution time for the ith wavefront. Then the execution time T for the whole program is $T = \sum_{i=1}^{n_w} T(w_i)$.

The time for each wavefront is estimated as the time required for all the tiles in one wavefront divided by the number of processors P. The execution time for each tile includes the time for fetching the data required to start the computation and the actual computation time. Here, we also assume that all the tiles are homogeneous, so the execution time for each tile is the same. Let $NTile(w_i)$ be the number of tiles in the w_ith wavefront, L_{mem} be the time spent for memory fetching, and T_{tile} be the computation time for each tile. Then

$$T(w_i) = (L_{mem} + T_{tile})\left(\frac{NTile(w_i)}{P}\right)$$

As we stated before, stencil computations become computation bound after tiling. We assume that the computation is large enough that once the computation starts, all the memory transfer can be overlapped by computation. Therefore, at each wavefront, we only have to pay the memory latency cost for the first tile to start the computation. Then

$$T(w_i) = L_{mem} + \frac{T_{tile}NTile(w_i)}{P}$$

which gives

$$\begin{aligned}
T &= \sum_{i=1}^{n_w} T(w_i) \\
&= \sum_{i=1}^{n_w}(L_{mem} + \frac{T_{tile}NTile(w_i)}{P}) \\
&= n_w L_{mem} + \sum_{i=1}^{n_w}\left(\frac{T_{tile}NTile(w_i)}{P}\right) \\
&\approx n_w L_{mem} + \frac{\text{total\_computation}}{P}
\end{aligned}$$

Where *total_computation* is the total number of computations in the program. For a given problem size and processor number, the part $\frac{\text{total\_computation}}{P}$ is invariant. For the rest of the analysis and the tile size selection, we will focus on the variant part $(n_w L_{mem})$.

As described in the wavefront code generation section, the number of wavefronts n_w can be computed using $n_w = \sum_{i=1}^{d} \left(\frac{N_i}{s_i}\right)$, where N_i is the size for the ith dimension, s_i is the tile size for ith dimension, and d is the number of dimensions that are tiled. However, for stencil computations, time skewing has to be applied to tile the whole computation space of stencil computations to enable tiling. Assume that dimension i is skewed with respect to dimension t, and the original size of dimension i is M_i and original size of dimension t is R_t. After skewing, the size of the skewed dimension i can be computed as $M_i + \alpha_i R_t$, α_i is called skewing factor for the ith dimension [2]. For example, α is 1 for Jacobi 1D for the data dimension. Therefore, the number of wavefronts can be computed as

$$n_w = \sum_{i=1}^{d} \left(\frac{M_i + \alpha_i R_t}{s_i}\right)$$

The memory latency is estimated by counting the volume of data needed to start the computation of a tile. For example, for Smith-Waterman, each tile requires the memory from the top boundary and left boundary. Therefore, the memory latency is $(s_i + s_j)$. But for 2D cases like Jacobi 2D, it requires faces from three directions, the faces above, the faces at left and the faces in front. For most 2D cases, the memory required for each tile can be represented as $\beta_1 s_i s_j + \beta_2 s_i s_t + \beta_3 s_j s_t$, where β_1, β_2 and β_3 are appropriate constants that can be automatically deduced – the number of faces required at each side.

Above, we described our analysis for standard wavefront parallelization. Now, we will extend it to our multi-pass parallelization strategy. The total number of computations for a given problem size remains invariant. And the memory latency for each wavefront is still modeled as the memory latency for one tile in the wavefront. The different part is the computation for the total number of wavefronts. The total number of wavefronts in the whole program is the number of wavefronts in each pass multiplied by the number of passes. Using the same idea of computing the number of tiles from Andonov [2]. The number of passes N_{pass} can be computed using the following

$$N_{pass} = \prod_{i=1}^{d'} \left(\frac{M_i + \alpha_i R_t}{s_i'}\right)$$

Where d' is number of dimensions that are tiled at the outer level, s_i' is the tile size for the outer level tiling. Since each pass can be viewed as a one level tiling problem. the number of wavefronts in each pass can be computed in the same way as computing the number of wavefronts in the whole program, but with the original size of each dimensions involves the pass sizes. For example, for a Smith-Waterman problem with problem size $M \times N$ and pass height H_j, the number of passes is $\frac{N}{H_j}$, the skewing factor is 0. Each pass can be viewed as a one level tiling problem, and the number of wavefronts for this tiling problem is $\frac{M}{s_i} + \frac{H_j}{s_j}$.

4.2 Memory analysis

In Section 3, we mathematically quantified the volume of off-chip memory transfer for Smith-Waterman. The main idea is counting the volume of boundary points for all the passes. This main idea remains the same when we move on to a higher number of dimensions and programs with more complicated dependences. Therefore, the volume of memory transfer is represented as

$$V = V_{surface} N_{pass}$$

where $V_{surface}$ is the volume of the surface area where memory access happens, and N_{pass} is the total number of passes. The number of passes is computed in the same way as we described in the execution time analysis.

Let's take the example shown in Figure 4. The problem size is N_i along the i dimension, N_j along the j dimension and R along the t dimension. The pass size is H_i along i dimension, and H_t along the t dimension. The corresponding inner tile sizes for each pass along each dimension is s_i, s_j and s_t.

To compute the values in a tube, it requires values from the bottom ij face, the left jt face and the front it face, since for stencil programs, the same memory loaded into a tile can be reused, the write back is not going to be a miss. Therefore, $V_{surface} = \beta_1 H_i N_j + \beta_2 H_t N_j + \beta_3 H_i H_t$. Based on the dependence of the computation, different number of faces or even different faces may be needed. The number of passes for the example in Figure 4 is simply computed as $N_{pass} = \frac{N_i}{H_i} \times \frac{R}{H_t}$. Therefore, the total volume of off-chip memory transfer is estimated as $V = \frac{N_i R}{H_i H_t}(H_i N_j + H_t N_j + H_i H_t) = \frac{R}{H_t}(N_i N_j) + \frac{N_i}{H_i}(R N_j) + N_i R$.

4.3 Trade-offs between Memory and execution time

The goal of our multi-pass strategy is to minimize the volume of off-chip memory access. However, this might bring in some extra overhead to the program. Let's take Smith-Waterman with problem size $M \times N$ as an example, the execution time is represented as $T = n_w L_{mem} + \frac{\text{total\_computation}}{P}$.

Since the part $\frac{total\_computation}{P}$ remains invariant for a given problem size and processor number, we only consider the term $n_w L_{mem}$. Next, we will quantify this part for both standard wavefront parallelization and our multi-pass strategy on this example.

With the standard tiling and wavefront parallelization, $n_w = \dfrac{M}{s_i} + \dfrac{N}{s_j}$, s_i and s_j are the tile sizes for the i and j dimension. For our multi-pass strategy, if we divide the computation into passes along the j dimension, the number of passes is $\frac{N}{H_j}$, where H_j is the pass size along j dimension. The number of wavefronts in each pass is $\dfrac{M}{s_i} + \dfrac{H_j}{s_j}$. Then the total number of wavefronts for the multi-pass strategy is $\dfrac{N}{H_j}(\dfrac{M}{s_i} + \dfrac{H_j}{s_j}) = \dfrac{NM}{H_j s_i} + \dfrac{N}{s_j}$. Since $M > H_j$, and if the same s_i and s_j is used for both strategy, the total number of wavefronts in our multi-pass strategy is greater than the total number of wavefronts in the standard wavefront parallelization. This will introduce more overhead for the memory latency part. Therefore, the tile sizes in the multi-pass strategy has to be tuned carefully that the overhead can be minimized with a certain gain in off-chip memory transfer.

Furthermore, it is known that time skewing introduces a pipeline filling up stage at the first few wavefront time step and a pipeline flushing stage at the last few time steps, and this fill-flush process introduces some overhead [20, 3]. The pipeline fill-flush overhead can be ignored for the standard wavefront parallelization strategy, since it only pays this overhead once. However, in our multi-pass strategy, we are paying this overhead at each pass, and this can potentially introduce overhead that can not be ignored. However, this can be mitigated by an improved code generation strategy, where the flush of a pass is overlapped with the fill of the next pass.

5 Experiments

To evaluate our cache-efficient parallelization strategy, we performed experiments on a set of benchmarks with stencil like dependences. In this section, we describe our experimental setup including detailed benchmark information, processor architecture, execution environment, etc. We also show the comparison between standard wavefront parallelization and our multi-pass parallelization strategy on execution time and off-chip memory traffic metrics.

5.1 Experimental setup

Our experiments are performed on two intel multicore processors: Intel Xeon E7-4830 and Intel Xeon E5-2620 v2. Table 1 describes the detailed hardware configuration for the two processors. Both processors have three levels of cache and the main characteristics of the cache hierarchy is outlined in Table 2. The main characteristics includes the type, size, line size and associativity for each level of cache. The L1 instruction cache is not included in the table, since we are interested in the data cache. The last level cache is shared among all the cores for both architectures.

Processor	Xeon E7-4830	Xeon E5-2620 v2
Architecture	Westmere	Ivy-Bridge
Clock speed	2.13 GHz	2.1 GHz
Number of cores	8	6
Level of cache	3	3

Table 1: Hardware specifications for Intel Xeon E7-4830 and Intel Xeon E5-2620 v2.

Processor	Cache level	Cache type	Cache size	Line size	Associativity
Xeon E7-4830	1 (private)	data	32 KB	64 B	8
	2 (private)	unified	256 KB	64 B	8
	3 (shared)	unified	24 MB	64 B	24
Xeon E5-2620 v2	1 (private)	data	32 KB	64 B	8
	2 (private)	unified	256 KB	64 B	8
	3 (shared)	unified	15 MB	64 B	20

Table 2: Cache hierarchy for Intel Xeon E7-4830 and Intel Xeon E5-2620 v2.

Above, we described the hardware specifications for both architectures. Next, we will describe the software configuration. Both platforms are running Linux, and all the programs are compiled using icc 12.1.2 with the optimization level -$O3$ and -*funroll-loops*. We use the last level cache misses as an estimation of the off-chip memory accesses, since each last level cache miss is going to cause an off-chip memory access. To measure the last level cache misses, we accesses the hardware counter through PAPI 3.5. Table 3 gives description about the main characteristics of each benchmark. Our benchmark suit includes stencil computations with different number of data dimensions, stencil orders, neighboring points, number of floating point operations per iteration and also different number of variables involved in the computation.

Benchmark	Data D	Order	NP	NV	FPI
Smith-Waterman	1	first	3	1	9
Jacobi 2D	2	first	5	1	5
Heat 2D	2	first	5	1	9
Seidel 2D	2	first	9	1	9
FDTD 2D	2	first	5	3	11
Wave 2D	2	third	13	1	13

Table 3: Details about the benchmarks. Data D is the number of dimensions of data grid, another time dimension is needed for computation. NP stands for neighboring points, it means the number of neighboring points needed for the computation of each point. NV represents the number of variables that have to be computed during the computation. FPI is floating point operations per iteration.

5.2 Tile size selection

Tile sizes have a direct impact to the execution time of the tiled code. In our experiments, we use the analysis described in Section 4 to guide the choice of tile sizes. As we described in Section 4.1, the amount of time spent on computation is the same for all different tile sizes with a given problem size, and the amount of data transferred to start the computation at each wavefront is different. Therefore, for the standard wavefront parallelization technique, we pick up the tile size that minimizes the memory transfer among all the tile sizes that fits into L2 cache. For our multi-level parallelization strategy, we compute the amount of off-chip memory accesses like we described in Section 4.2 and the data transferred to start the computation. Then we pick up some points on the pareto frontier, and measured both last level cache misses and execution time for each point.

5.3 Result Analysis

To demonstrate that our code generator generates efficient code, we compared the execution timeof standard wavefront parallelization code generated by our Parameterized Hierarchy Tiled (PHTile) code generator with the code generated by Pluto [6] (a tiled code generator with fixed tile size) and Pochoir [34] (cache oblivious tiled code generator) . The tile sizes picked up for our PHTile is passed into Pluto to generate tiled code. The result is shown in Figure 6 and 7. The base line used in the execution timecomparison for each program is the execution time of the best performed generated program on one thread. For all the benchmarks, the code generated by our PHTile gets comparable execution time with Pluto and Pochoir on both platforms.

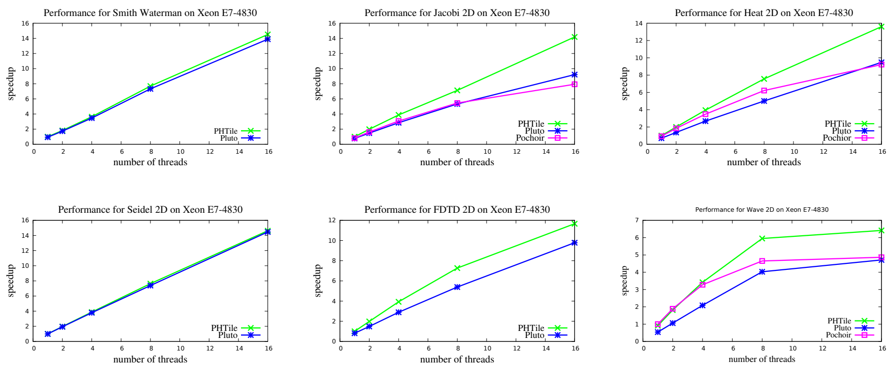

Figure 6: Execution time comparison for standard wavefront parallelization from PHtile with Pluto and Pochoir on Intel Xeon E7-4830. There is no execution time for Pochoir for Smith-Waterman, Seidel and FDTD. This is because Pochoir requires the neighboring points are all from the previous time step, but Smith-Waterman, Seidel and FDTD examples need points from the current time step.

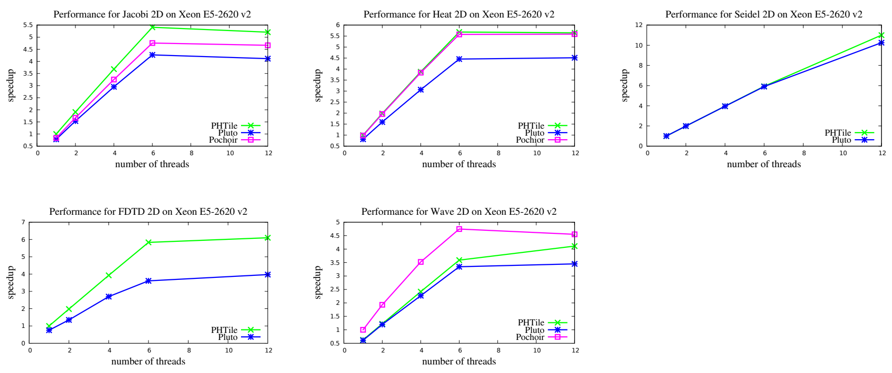

Figure 7: execution time comparison for standard wavefront parallelization from PH-Tile with Pluto and Pochoir on Intel Xeon E5-2620 v2.

In Figure 8 and Figure 9, we show the comparison result of execution timeand last level cache miss between the last level cache misses and execution time on both platforms. On Intel Xeon E7-4830 For FDTD 2D, Heat 2D and Seidel 2D, more than 40 times reduction on the last level cache misses can be achieved with only about 10% slow down in performance. More reductions can be achieved with more degeneration of performance. And for Jacobi 2D and Wave 2D, we can get about 30-fold reduction in last level cache misses with about 30% slow down in performance. For Smith-Waterman, we even got up to 7000 times improvement on last level cache misses. According to our previous analysis on Smith-Waterman, the improvement of cache misses we can get is $\frac{MH}{M+H} \times \frac{x+y}{xy}$, plug in the tile sizes we selected for Smith-Waterman, the cache miss improvement we got can reach under the most idea situation is about 10^4. Similarly, on Xeon E5-2620 v2, with only 10% slow-down in performance, we get up to 50-fold reduction in last level cache misses for Jacobi 2D, 30-fold for FDTD 2D and Wave 2D and up to 300-fold reduction for Heat 2D and Seidel 2D. There is also a three orders of magnitude reduction in last level cache misses for Smith-Waterman on Xeon E5-2620 v2.

As we noticed from the result of two platforms, a better last level cache miss reduction can be achieved on Xeon E5-2620 v2 compared with Xeon E7-4830. This is probably because the Xeon E5 has a smaller last level cache with lower number of associativity, and this can potentially increase the chance of capacity cache misses and conflict cache misses. And our strategy work even better under those situation.

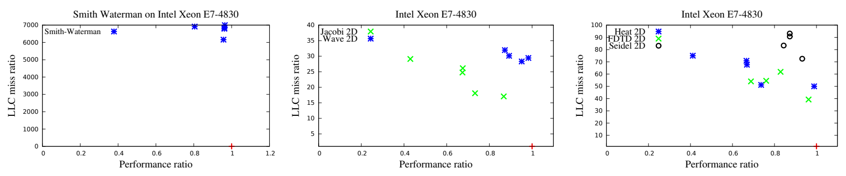

Figure 8: Last level cache (LLC) miss improvement for different benchmarks on Intel Xeon E7-4830. The y axis is LLC ratio, which is computed as the ratio of the measured LLC misses for our multi-pass strategy to the LLC misses of the standard wavefront parallelization. Similarly, the performance ratio along x axis represents the ratio of the execution time of our multi-pass strategy to the best standard wavefront performance. The red point in each figure represents the position for standard wavefront parallelization.

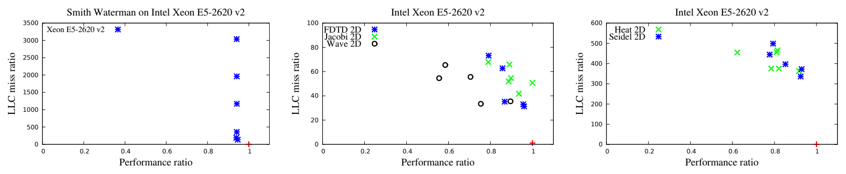

Figure 9: Last level cache (LLC) miss improvement for different benchmarks on Intel Xeon E5-2620 v2.

6 Related Work

Many authors have worked on optimizing stencil computations for improving locality for stencil computations, in both, shared memory and distributed memory or hybrid platforms. Dursun et at. [9] and Peng et al.[26] apply techniques like cache blocking and register blocking on stencil computation. These techniques divide the computation grid into small blocks so that each block fits into the cache or register. Further benefits have been showed by applying those techniques on multiprocessors. However, the amount of cache reuse is very limited with those techniques, since all the points in the data grid still have to be read once at every time step when the data grid size is large enough.

Time skewing [37, 8] is one of the most important approaches that exploit data locality across multiple time steps. It looks at the dependencies in the whole iteration domain, skews it with respect to the time dimension, then divides it into rectangular tiles. Wonnacott also generalized this this idea to handle imperfectly nested loops in the context of the Omega tool. The same effect of time skewing can also be achieved by hyperplane tiling [15, 6, 3]. Instead of skewing the computation space to make rectangular tiling legal, they are trying to find a legal hyperplane to cut the whole computation space. Due to dependences between tiles, most authors [37, 6] subsequently parallelize the tiled program with wavefront parallelization. As we described in section 3, the off-chip memory transfer is still significant with large problem size, since the potential of data reuse between successive wavefronts is not exploited. In this paper, we achieved this by adding an additional level of tiling at the "outer" or "pass" level.

Strzodka present a technique called cache accurate time skewing (CATS) [32] for stencil computations. CATS is based on the reduction of higher dimensional problem into lower dimensional, non-hierarchical problem. In CATS, a subset of the dimensions are tied to form large tiles, and a sequential wavefront traversal is performed inside the tiles and the parallelism is explored among the tiles. To obtain the maximally concurrent start, the tile shape is chosen such that the parallel dimension and the tiled dimension forms a diamond shape. They also present a technique for choosing the tiling parameters optimally: they maximize the tile size, bounded by the size of the private (L2) cache of one thread, such that it supports reuse inside a tile. In CATS, the overall tiling strategy is such that each diamond shaped tile accesses distinct values, and so there is no reuse between tiles. They do not address energy minimization, so they use the private, rather than the last level cache as capacity bound. However, we can easily adapt their method to optimize for last-level cache misses. Using the model of Section 4 we can show that their strategy for optimal tile size, adapted to the last level rather than private cache would nevertheless perform $\sqrt{P}$ more memory accesses than ours, where P is the number of processors. Of course these are only analytical predictions, and more detailed experimental comparison would be needed.

Frigo and Strumpen [10] present a cache oblivious algorithm [28] for stencil computations using tiles with trapezoidal surfaces. They recursively tile the whole computation space into small tiles that fit into a given cache. However, their algorithm is not designed to make use of the parallel architecture. In the later work [31, 34], the parallelism is achieved by an improved space cut such that independent tiles can be obtained. Frigo and Strumpen showed that for an ideal cache with size Z, the cache oblivious strategy can save a factor of $\Theta(Z^{\frac{1}{n}})$ compared with the naive implementation. However, the reuse among parallel tiles is still not explored. There are also some other factors that can impact the efficiency of the cache-oblivious algorithm, e.g., without knowing the cache size, bad choice of base case can be made and it can significantly increase the overall cache misses.

7 Conclusion

We presented an cache-efficient parallelization strategy for dense stencil like programs, on which polyhedral analysis can be used and tiling can be applied. Our strategy first divides the computations into passes. Since a pass is nothing but a very large atomic tile, one may view the strategy as hierarchical tiling. Hierarchical tiling can be applied at multiple levels. Although we did not do so in this paper, an additional finer grain of parallelism could (and should) be applied to exploit vector-level parallelism.

We mathematically analyzed the memory behavior and performance for both the standard wavefront parallelization strategy and our strategy. We showed the possible trade-offs between the execution time and off-chip memory traffic. We also developed a tiled code generator for our strategy. Our experimental results showed that, under most situations, a significant reduction in off-chip memory traffic can be achieved with very modest slowdown.

There are two main reasons for the slowdown. First is the the latency at the start of each wavefront—our multi-pass strategy executes many more wavefronts than the standard one. Moreover, we pay the price of pipelining fill-flush for wavefront parallelization of *each pass*. Both of these can be easily overcome. First, a "tile-level" prefetch can be used to anticipate the data required by the first tile in the next wavefront. Second, inter-pass overlapping can be added to the code that is generated. As we add these capabilities to the code generator, we expect that the benefits such as energy gains will essentially come "for free."

In the long term, there are a number of open questions. The quantitative model we proposed is approximate, and the experiments further approximated it by counting just the number of cache misses. Many off-chip transfers, e.g., accesses to different values within the same cache line and write-back transfers, do not provoke a cache miss. Second, the model does not account for conflict misses. Therefore, an interesting problem that we are currently addressing is to precisely model the number of transfers. Furthermore, We did not completely validate the accuracy of the cost model for a wide range of parallelization parameters. We simply illustrated in our experiments, the fact that huge savings are possible. We plan to validate the model, and then use it to precisely formulate a set of Pareto optimal design points and then move from a "user-guided" code generator to a fully automatic compilation system. This would involve automatically determining the parallel schedule, the skewing factors for each dimension, and automatic detection of the tilable band and setting all the tile size parameters at all levels.

References

[1] A.M. Aji and Wu chun Feng. Optimizing Performance, Cost, and Sensitivity in Pairwise Sequence Search on a Cluster of Playstations. In *8th IEEE International Conference on BioInformatics and BioEngineering (BIBE), 2008.*, pages 1–6, 2008.

[2] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal Semi-Oblique Tiling. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '01, New York, NY, USA, 2001. ACM.

[3] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling Stencil Computations to Maximize Parallelism. In *the International conference on high performance computing, networking, storage and analysis*,

SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[4] Cedric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Cedric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[7] Bo Chen, Yun Xu, Jiaoyun Yang, and Haitao Jiang. A New Parallel Method of Smith-Waterman Algorithm on a Heterogeneous Platform. In *Proceedings of the 10th international conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] D. Wonnacott. Time Skewing for Parallel Computers. In *the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 477–480, London, UK, 2000. Springer-Verlag.

[9] Hikmet Dursun, Ken-ichi Nomura, Weiqiang Wang, Manaschai Kunaseth, Liu Peng, Richard Seymour, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. In-Core Optimization of High-Order Stencil Computations. In Hamid R. Arabnia, editor, *PDPTA*. CSREA Press, 2009.

[10] M. Frigo and V. Strumpen. Cache Oblivious Stencil Computations. In *International Conference on Supercomputing (ICS), 2005.*, pages 361–366, Cambridge, MA, June 2005.

[11] Robert M. Haralick and Linda G. Shapiro. *Computer and Robot Vision*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1992.

[12] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric Multi-Level Tiling of Imperfectly Nested Loops. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, New York, NY, USA, 2009. ACM.

[13] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric Multi-Level Tiling of Imperfectly Nested Loops. In

Proceedings of the 23rd international conference on Supercomputing, ICS '09, New York, NY, USA, 2009. ACM.

[14] M.D. Hill and A.J. Smith. Evaluating Associativity in CPU Caches. *Computers, IEEE Transactions on*, 38(12), 1989.

[15] F. Irigoin and R. Triolet. Supernode Partitioning. In *the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

[16] S. Kamil, Cy Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010.

[17] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The Organization of Computations for Uniform Recurrence Equations. *J. ACM*, 14(3), July 1967.

[18] Daegon Kim. *Parameterized and Multi-Level Tiled Loop Generation*. PhD thesis, Colorado State University, Fort Collins, CO, USA, 2010.

[19] C. T. King, W. H. Chou, and L. M. Ni. Pipelined Data Parallel Algorithms-II: Design. *IEEE Trans. Parallel Distrib. Syst.*, 1(4), October 1990.

[20] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM.

[21] Yang Liu, Wayne Huang, John Johnson, and Sheila Vaidya. GPU Accelerated Smith-Waterman. In *International Conference on Computational Science (4)*, Lecture Notes in Computer Science. Springer, 2006.

[22] Andreas Merkel and Frank Bellosa. Memory-Aware Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, Berkeley, CA, USA, 2008. USENIX Association.

[23] Lauri Minas and Brad Ellison. The Problem of Power Consumption in Servers. *Intel Press*, 2009.

[24] D.I. Moldovan and J. A B Fortes. Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays. *IEEE Transactions on Computers*, C-35(1):1–12, Jan 1986.

[25] D.J. Palermo, E. Su, J.A. Chandy, and P. Banerjee. Communication Optimizations Used in the Paradigm Compiler for Distributed-Memory Multicomputers. In *Parallel Processing, 1994. Vol. 1. ICPP 1994. International Conference on*, volume 2, 1994.

[26] Liu Peng, R. Seymour, K. Nomura, R.K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W.R. Volz, and C.C. Wong. High-Order Stencil Computations on Multicore Clusters. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2009.*, may 2009.

[27] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. *Combined Iterative and Model-Driven Optimization in an Automatic Parallelization Framework*. IEEE Computer Society Press, New Orleans, LA, 2010.

[28] Harald Prokop. Cache-Oblivious Algorithms. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1999.

[29] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye. Towards Optimal Multi-Level Tiling for Stencil Computations. In *International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007.*, IPDPS '07, 2007.

[30] Aaron Sawdey, Matthew O'Keefe, Rainer Bleck, and Robert W. Numrich. The Design, Implementation, and Performance of a Parallel Ocean Circulation Model. In *the sixth ECMWF workshop on the use of parallel processors in meteorology*, pages 523–550, 1994.

[31] R. Strzodka, M. Shaheen, D. Pajak, and H-P. Seidel. Cache Oblivious Parallelograms in Iterative Stencil Computations. In *24th ACM/SIGARCH International Conference on Supercomputing (ICS)*, pages 49–59, Tsukuba, Japan, June 2010.

[32] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache Accurate Time Skewing in Iterative Stencil Computations. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. IEEE Computer Society, September 2011.

[33] A. Taflove and S. C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech Hourse Publishers, third edition, 2005.

[34] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir Stencil Compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, New York, NY, USA, 2011. ACM.

[35] Colorado State University. HiTLoG: Hierarchy tiled loop generator. http://www.cs.colostate.edu/MMAlpha/tiling/.

[36] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[37] David Wonnacott. Achieving Scalable Locality with Time Skewing. *Int. J. Parallel Program.*, 30(3):181–221, Jun 2002.

[38] Jingling Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[39] Jianhui Yue, Yifeng Zhu, and Zhao Cai. An Energy-Oriented Evaluation of Buffer Cache Algorithms Using Parallel I/O Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1565–1578, 2008.