Hierarchical Overlapped Tiling

Xing Zhou, Jean-Pierre Giacalone[†], María Jesús Garzarán,
Robert H Kuhn[†], Yang Ni[†], David Padua

University of Illinois at Urbana-Champaign
{zhou53,garzaran,padua}@illinois.edu

[†]Intel Corporation
{jean-pierre.giacalone,bob.kuhn,yang.ni}@intel.com

ABSTRACT

This paper introduces hierarchical overlapped tiling, a transformation that applies loop tiling and fusion to conventional loops. Overlapped tiling is a useful transformation to reduce communication overhead, but it may also generate a significant amount of redundant computation. Hierarchical overlapped tiling performs overlapped tiling hierarchically to balance communication overhead and redundant computation, and thus has the potential to provide better performance.

In this paper, we describe the hierarchical overlapped tiling optimization and its implementation in an OpenCL compiler. We also evaluate the effectiveness of this optimization using 8 programs that implement different forms of stencil computation. Our results show that hierarchical overlapped tiling achieves an average 37% speedup over traditional tiling on a 32-core workstation.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Languages, Performance

Keywords

Loop tiling and fusion, compiler optimization, stencil computation

1. INTRODUCTION

1.1 Overlapped Tiling

Numerous techniques for the tiling of iteration spaces have been proposed. The goal of tiling is to for improve data locality [27, 28, 1, 2, 24, 9, 22], or contribute to the scheduling of parallel computation [26, 29, 6, 10, 30]. A complementary transformation, loop fusion can be used to decrease loop overhead and enhance locality [17, 20].

In this paper, we discuss transformation by tiling and fusion of stencil computation. Consider the code in Figure 1-(a). Although this figure shows a natural representation of the computation, the pair of loops may cause unnecessary cache misses, depending on how they are scheduled. If the loops are scheduled in the order specified by the code, the second loop will incur frequent cache misses. To increase locality, and also coarsen the granularity of the parallel tasks, the programmer can tile and fuse the loops, as shown in Figure 1-(b). The resulting code requires an explicit barrier to guarantee correctness, because of the data dependences between neighboring tiles of iterations (during iteration t of the outer loop, j consumes data produced by adjacent tiles of loop i, namely tiles $t-1$ and $t+1$ or just one of them at the boundaries). Notice that locality would improve if the same task executes the corresponding i and j tiles in the code of Figure 1-(b). However, good locality is only possible if array A can be kept in cache memory when the execution moves from the first to the second loop. If, however, the array A is larger than the total cache of the processors executing the loops, the traditional loop fusion and tiling transformation applied in Figure 1-(b) will not benefit from locality, because all the iterations of the i loop must complete before the j loop executes. Besides the difficulties for achieving locality of naive tiling, the parallelization transformation may do a suboptimal job because of the barrier introduced. On some architectures, barriers are expensive synchronization operations, and could additionally cause load imbalance. Furthermore, the transformation from Figure 1-(a) to Figure 1-(b) is not possible in some languages, such as OpenMP and OpenCL [14], which do not allow global barriers inside data parallel constructs.

To remove the synchronization and enhance locality, the code can be transformed into the form shown in Figure 1-(c). In this case, each iteration of the outer loop t produces all the data it needs so that its iterations (which correspond to tiles) are independent from each other. This is achieved because each iteration performs redundant computation. The result is a code without the BARRIER and with increased locality.

In the example in Figure 1-(c) loop i produces $A[max(0, t * T - 1) : min(N, (t + 1) * T)]$ in each iteration of the outer

```
parallel for(int i = 0 : N-1)
    A[i] = ...;
parallel for(int j = 0 : N-1)
    ... = A[j-1] + A[j] + A[j+1];
```

(a) Original loops

```
parallel for(int t = 0 : N/T)
    for(int i = t*T; i < min(N, (t+1)*T; i++)
        A[i] = ...;
    BARRIER;
    for(int j = t*T; j < min(N, (t+1)*T; j++)
        ... = A[j-1] + A[j] + A[j+1];
}
```

(b) Traditional tiling and fusion

```
parallel for(int t = 0 : N/T) {
    for(int i = max(0,t*T-1);
            i < min(N,(t+1)*T+1); i++)
        A[i] = ...;
    for(int j = t*T; j<min(N,(t+1)*T); j++)
        ... = A[j-1] + A[j] + A[j+1];
}
```

(c) Overlapped tiling

Figure 1: A simple tiling example for parallel loops

loop, that is, $T+2$ elements, 2 more than the number of elements of A computed by loop i in Figure 1-(b). In total the N/T executions of loop i in Figure 1-(c) produce $\frac{T+2}{T} * N$ elements, so this loop performs $\frac{T+2}{T} * N - N = \frac{2*N}{T}$ more iterations than the corresponding loop of Figure 1-(b). We call the transformation leading to a loop of the form of Figure 1-(c) *overlapped tiling*.

Figure 2-(a) shows a code snippet which represents a typical stencil computation. Pairs of consecutive executions of the inner loop form a pattern similar to that of the two inner loops in Figure 1-(a). If we apply overlapped tiling repetitively and fuse all K executions of the inner loop, it is possible to execute the outer loop without using any barrier. The number of consecutive loops fused is the *depth* of the transformation. In this example, the fusion *depth* is K. The total amount of redundant computation usually grows with the value of *depth*. Figure 2-(b) shows the area of overlap. The triangles that bracket each tile represent the redundant computation.

1.2 Hierarchical Overlapped Tiling

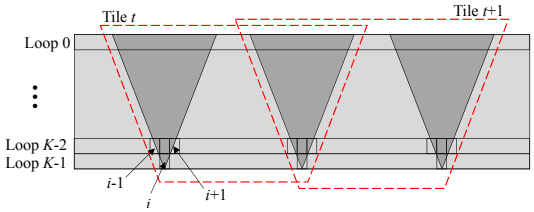
While overlapped tiling removes synchronization and enhances locality, it usually suffers from substantial amount of redundant computation. As the fusion *depth* increases, more synchronizations can be removed, but there is also an increase in the total amount of redundant computation (the shadowed triangle areas in Figure 2-(b)). Hence, we propose the use of *hierarchical overlapped tiling* to balance communication overhead and redundant computation. Figure 3 contrasts overlapped tiling with hierarchical overlapped tiling.

The example assumes 8 consecutive loops executing on a 4-way/8-core multicore system where the two cores on each processor share the last level cache. Figure 3-(a) shows the result of overlapped tiling across the eight cores while Figure 3-(b) shows the result of applying overlapped tiling hierarchically. In Figure 3-(b), overlapped tiling is first applied across the four processors. Within each processor, pairs of consecutive loops are fused. This forces a local barrier between each pair of loops ($loop_0$ and $loop_1$, $loop_2$ and $loop_3$, and so on). This barrier, however, only synchronizes the two cores on each processor and therefore its cost should be relatively low. Within each processor, overlapped tiling is applied to enable the parallel execution of pairs of loops across the two cores without the need for a barrier. Compared to overlapped tiling, the total amount of redundant computation (the shadowed triangle areas between different processors and the smaller shadowed triangles between neighboring cores) caused by the 2 levels of tiling is much smaller. This reduction of the redundant computation is the main source of the performance benefit of hierarchical overlapped tiling over plain overlapped tiling.

In this paper we describe the compiler implementation of the hierarchical overlapped tiling optimization. Automating

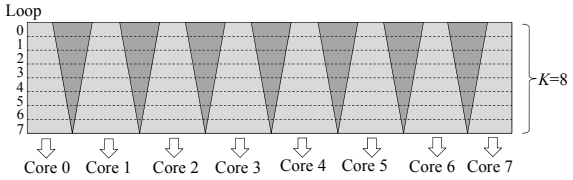
```
for(int k = 0 ; k < K; k++) {
    parallel for(int i = 0 : N-1)
        B[i] = A[i-1] + A[i] + A[i+1];
    swap(A, B);
}
```

(a) Code snippet of K consecutive loops in a stencil code

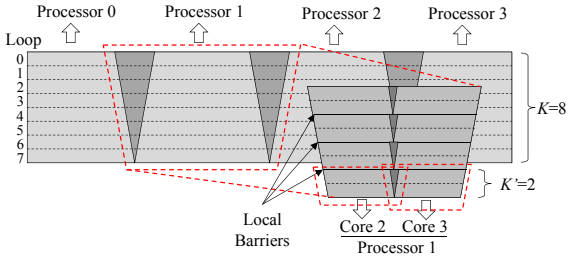


(b) Overlapped tiling

Figure 2: Overlapped tiling of K loops



(a) Overlapped tiling



(b) 2-level hierarchical overlapped tiling

Figure 3: Comparison of overlapped tiling and hierarchical overlapped tiling on a 4-way/8-core multicore system, where each processor contains 2 cores on chip that share the last level cache.

the transformation in a compiler simplifies the task of the programmer during code development and porting. This paper makes the following contributions:

- The introduction of a new tiling transformation called *hierarchical overlapped tiling*.
- The description of its implementation in an OpenCL compiler with the support of Cetus [12] and the Omega library [13] for OpenCL programs which implements the proposed transformation.
- An evaluation of the effect of the techniques on stencil computation. Our experimental results show that hierarchical overlapped tiling is effective and achieves significant speedups when compared to the traditional loop fusion and tiling transformation.

The rest of the paper is organized as follows: Section 2 gives a quantitative analysis of overlapped and hierarchical overlapped tiling; Section 3 describes our compiler implementation; Section 4 discusses the environmental setup for the experimental evaluation; Section 5 shows our experimental results; Section 6 discusses our related work; Section 7 presents the conclusions.

2. ANALYTICAL MODELING

This section gives a quantitative analysis of overlapped tiling and hierarchical overlapped tiling.

2.1 Integer Tuple Set and Relation

We use the notion of iteration space to describe and analyze the transformation introduced here. A d-dimensional *integer tuple* $\vec{x} = [x_0, x_1, ..., x_{d-1}]$ is a vector of d integers. Constraints in the form of equations and inequalities can be used to describe sets of integer tuples. For example, the set $S_0 = \{[1,1],[1,2],...,[1,N]\}$ can be represented as $\{[i,j] : i = 1 \wedge 1 \leq j \leq N\}$.

We assume that the arithmetic expressions in the equations and inequalities are affine and the terms are integers. Logical operators $\neg$, $\wedge$ and $\vee$, and the existential and universal quantifiers $\exists$ and $\forall$ are also needed to describe the set of integer tuples. The representations we use are known as Presburger formulas [15]. In our implementation, these expressions are manipulated using the Omega Library [13].

We also represent *integer tuple relation*s with rules described by Presburger formulas. For example the relation $T = \{[i,j] \rightarrow [x,y] : i-1 \leq x \leq i \wedge j-1 \leq y \leq j+1\}$ when applied to the previous S_0 yields the following set:

$$T(S_0) = \{[x,y] : 0 \leq x \leq 1 \wedge 0 \leq y \leq N+1\}$$

Note that given a relation R the size and shape of the original set S and that of $R(S)$ for a relation R can be different. The union $\cup$ of two relations T_1 and T_2 is defined as:

$$T_1 \cup T_2 = T, if\ \forall S, T_1(S) \cup T_2(S) = T(S)$$

2.2 Terminology

Consider K consecutive parallel loops $loop_0$, $loop_1$, ..., $loop_{K-1}$. Assume that the k-th loop $loop_k$ $(0 \leq k < K)$ has the form shown Figure 4.

```
parallel for(int i⃗ = [i_0, i_1, ..., i_{D-1}] ∈ I_0) { // loop_0
    ...
}
parallel for(int i⃗ = [i_0, i_1, ..., i_{D-1}] ∈ I_1) { // loop_1
    ...
}
    ...
parallel for(int i⃗ = [i_0, i_1, ..., i_{D-1}] ∈ I_k) { // loop_k
    ... = A_k^0[f_k^0(i⃗)];
    ... = A_k^1[f_k^1(i⃗)];
            ...
    ... = A_k^{L_k-1}[f_k^{L_k-1}(i⃗)];
    B_k^0[g_k^0(i⃗)] = ...;
    B_k^1[g_k^1(i⃗)] = ...;
            ...
    B_k^{M_k-1}[g_k^{M_k-1}(i⃗)] = ...;
}
    ...
parallel for(int i⃗ = [i_0, i_1, ..., i_{D-1}] ∈ I_{K-1}){// loop_{K-1}
    ...
}
```

Figure 4: A sequence of K parallel loops

Without loss of generality, we assume the body of $loop_k$ contains read access to L_k D'-dimensional arrays A_k^0, A_k^1, ..., $A_k^{L_k-1}$ and write access to M arrays B_k^0, B_k^1,...,$B_k^{M_k-1}$ which are also D' dimensional. We assume that no A array overlaps with a B array. To simplify discussion we assume $A_k^0 = A_k^1 = ... = A_k^{L_k-1} = A_k$ and $B_k^0 = B_k^1 = ... = B_k^{M_k-1} = B_k$. There are L_k references to A_k on the RHS of the first L_k assignment statements in the body of the loop: $A_k[\vec{f}_k^0(\vec{i})]$, $A_k[\vec{f}_k^1(\vec{i})]$, ..., $A_k[\vec{f}_k^{L_k-1}(\vec{i})]$, with $\vec{f}_k^l : \mathcal{Z}^D \rightarrow \mathcal{Z}^{D'}, 0 \leq l < L_k$. There are M_k references to elements of B_k on the LHS of the last M_k statements: $B_k[\vec{g}_k^0(\vec{i})]$, $B_k[\vec{g}_k^m(\vec{i})]$, ..., $B_k[\vec{g}_k^{M_k-1}(\vec{i})]$, with $\vec{g}_k^m : \mathcal{Z}^D \rightarrow \mathcal{Z}^{D'}, 0 \leq m < M_k$.

We compute 3 sets for $loop_k$: I_k, the iteration space; R_k, the set of subscripts of A_k; and W_k the set of subscripts of B_k:

$$R_k = \{\vec{r}\} = \bigcup_{l=0}^{L_k-1} \{[r_0, r_1, ..., r_{D'-1}] : \vec{r} = \vec{f}_k^l(\vec{i}) \wedge \vec{i} \in I_k\}$$

$$W_k = \{\vec{w}\} = \bigcup_{m=0}^{M_k-1} \{[w_0, w_1, ..., w_{D'-1}] : \vec{w} = \vec{g}_k^m(\vec{i}) \wedge \vec{i} \in I_k\}$$

We define C_k for the *consuming relation* as the relation from I_k to R_k, $C_k(I_k) = R_k$, and P_k for the *producing relation*, as the relation from W_k to I_k, $P_k(W_k) = I_k$. C_k and P_k represent the access pattern of the loop body. We use C_k and P_k to describe the different tiling transformations.

2.3 Overlapped Tiling

In this subsection we describe the overlapped tiling loop transformation.

Performing loop fusion and tiling for a sequence of loops of the form shown in Figure 4 is equivalent to finding Q parti-

tions (tiles) I_k^0, I_k^1, ..., I_k^{Q-1} of the iteration space I_k of each $loop_k$. Similar to the definition of R_k and W_k discussed in the last subsection, we define R_k^q as the set of array element indices of A for tile I_k^q, and W_k^q to denote the set of array elements indices of B for tile I_k^q. So, we have:

$$R_k^q = C_k(I_k^q), \qquad I_k^q = P_k(W_k^q) \;\; or \;\; W_k^q = P_k^{-1}(I_k^q)$$

Traditional loop fusion and tiling, such as the code shown in Figure 1-(b), produce orthometric tiles. Since the tiles are a partition of the iterations space, we have:

$$I_k^{q1} \cap I_k^{q2} = \phi, \quad q1 \neq q2$$

However, with overlapped tiling the sum of the size of the tiles I_k^q can be larger than the size I_k. We define the difference as the amount of redundant computation RC_k performed by $loop_k$:

$$RC_k = |I_k^0| + |I_k^1| + ... + |I_k^{Q-1}| - |I_k| \geq 0$$

In traditional loop fusion and tiling, the tiles of the different loops can be unrelated thanks to the barriers. However, in overlapped tiling the data read in an iteration tile must be produced within the same tile to eliminate the need of synchronization. To simplify the discussion, we assume that the data flow in the sequence of fused loops $loop_0$, $loop_1$, ... $loop_{K-1}$ forms a linear chain, which means that $A_{k+1} = B_k$ for all k. Under this assumption, it is necessary that $R_{k+1}^q \subseteq W_k^q$ in overlapped tiling and to avoid unnecessary work we set $R_{k+1}^q = W_k^q$. Hence the corresponding tiles I_{k+1}^q and I_k^q of neighboring loops are related as follows:

$$R_{k+1}^q = C_{k+1}(I_{k+1}^q) = P_k^{-1}(I_k^q) = W_k^q \quad \Rightarrow$$

$$I_k^q = P_k(W_k^q) = P_k(R_{k+1}^q) = P_k(C_{k+1}(I_{k+1}^q)) \qquad (1)$$

Equation 1 shows that the tiles of $loop_k$ are determined by the tiles of $loop_{k+1}$. Equation 1 provides the procedure to perform overlapped tiling: given an arbitrary partition of tiles for the last loop $loop_{K-1}$, the tiles of previous loops $loop_k$ $(0 \leq k < K)$ can be determined iteratively; then all the corresponding tiles from the different loops are fused to build the new loop body.

Consider the code in Figure 2-(a) as an example to perform overlapped tiling. After unrolling, there will be a sequence of K loops. Since every $loop_k$ in Figure 3-(a) is the same, we have:

$$I_0 = I_1 = ... = I_{K-1} = I = \{[i] : 0 \leq i < N\}$$
$$C_0 = C_1 = ... = C_{K-1} = C = \{[i \rightarrow x] : i - 1 \leq x \leq i + 1\}$$
$$P_0 = P_1 = ... = P_{K-1} = P = \{[x \rightarrow i] : i = x\}$$

Initially, we assign the following tiling partition for the last $loop_{K-1}$:

$$I_{K-1}^q = \{[i] : q \times N/Q \leq i < (q+1) \times N/Q\}$$

which simply evenly partitions the iteration space, I_{K-1}, where the size of each tile is $|I_{K-1}^q| = N/Q$.

Next, the previous loops can be tiled using Equation 1 (to simplify the discussion, the boundaries are ignored):

$$\begin{aligned} |I_{K-2}^q| &= |P_{K-2}(C_{K-1}(I_{K-1}^q))| = |P(C(I_{K-1}^q))| \\ &= |\{[i] : q \times N/Q - 1 \leq i < (q+1) \times N/Q + 1\}| \\ &= N/Q + 2 = |I_{K-1}^q| + 2 \\ |I_{K-3}^q| &= |P(C(I_{K-2}^q))| = N/Q + 4 = |I_{K-2}^q| + 2 \\ &... \\ |I_0^q| &= |P(C(I_1^q))| = N/Q + 2 \times (K-1) = |I_1^q| + 2 \end{aligned}$$

Therefore:

$$|I_k^q| = |P(C(I_{k+1}^q))| = N/Q + 2 \times (K - 1 - k)$$
$$RC_k = (\sum_{q=0}^{Q-1} |I_k^q|) - |I_k| = 2 \times Q \times (K - 1 - k)$$

The total amount of redundant computation RC is defined as the sum of the redundant computation of each $loop_k$:

$$RC = \sum_{k=0}^{K-1} RC_k = Q \times K \times (K-1) \qquad (2)$$

RC is a monotonic function of the number of tiles Q and the number of loops to fuse K. According to Equation 2, for the example shown in Figure 2-(a), we can see the trend: the amount of redundant computation RC increases with both Q and K. Although this observation is derived from the specific example, it is easy to see that the trend is true in general. Compared with traditional loop fusion and tiling, where synchronization is necessary, overlapped tiling saves the overhead of $K-1$ barriers. Suppose the average overhead of each barrier is t_s, and the average computation time for each iteration in the original code is t_c, the overhead difference between overlapped tiling over traditional tiling is:

$$\Delta Overhead = t_s \times (K-1) - t_c \times RC/Q \qquad (3)$$

2.4 Hierarchical Overlapped Tiling

According to the analysis in the last subsection, coarse grain tiles (and thus small number of tiles) reduce the amount of redundant computation in overlapped tiling. However, too few tiles would reduce the amount of parallelism. Similarly, reducing the number of fused loops also reduces the amount of redundant computation, but at the expense of the additional synchronization required between loops.

The goal of hierarchical overlapped tiling is to adapt to the memory hierarchy of the target machine. Consider a multicore system with N_p processors, where each processor has N_c cores sharing the last level cache. It is expected that the average overhead of synchronization of processors sharing a cache (t_s') will be significantly smaller than that of synchronizing cores on different processors (t_s) that need to communicate through main memory and/or bus.

$$t_s / t_s' \gg 1 \qquad (4)$$

Based on the above observation, we proceed as follows: first, we perform coarse-grain tiling for processors; then perform overlapped tiling within the tile that is assigned to each

processor, and generate sub-tiles for each core of the processor. During this 2-level tiling, the parameters for overlapped tiling are determined separately for each level.

For simplicity, we assume that the total number of tiles, Q, generated by the plain overlapped tiling discussed in the previous subsection is equal to the number of cores: $Q = N_p \times N_c$. During the first level tiling of the hierarchical overlapped tiling, only N_p tiles are generated, so the total amount of redundant computation introduced in this level is:

$$RC_1 = RC(N_p, K) = N_p \times K \times (K-1)$$

After the first level tiling, each processor q is assigned a coarse-grain tile: I_0^q, I_1^q, ..., I_{K-1}^q $(0 \leq q < N_p)$. The tile for each processor is implemented as a sequence of inner-loops, whose boundaries are defined by the constraints discussed at the beginning of Section 2.1. Then, overlapped tiling can be applied within each tile. However, since the sub-tiles are going to be mapped to cores of the same processor, it is expected that the overhead of synchronization of cores within the same processor, t_s' will be significantly smaller than the synchronization between cores across processors, t_s. As a result, for each tile in the second level of tiling the number of fused loops (fusion *depth*) can be smaller. Although this increases the amount of synchronization, it also reduces the amount of redundant computation. Suppose the second level of tiling only fuses K' consecutive loops each time and $K' < K$, then the amount of redundant computation for each processor in the second level tiling would be:

$$RC_2 = RC(N_c, K') = N_c \times K' \times (K'-1) \qquad (5)$$

Therefore, the total amount of redundant computation of 2-level overlapped tiling is:

$$\begin{aligned} RC' &= RC_1 + N_p \times RC_2 \\ &= N_p \times K \times (K-1) + N_p \times N_c \times K' \times (K'-1) \\ &= \frac{Q}{N_c} \times K \times (K-1) + Q \times K' \times (K'-1) \\ &= Q \times K \times (K-1) \times (\frac{1}{N_c} + \frac{K' \times (K'-1)}{K \times (K-1)}) \\ &\approx RC \times (1/N_c + (K'/K)^2) \end{aligned}$$

Furthermore, additional K/K' local synchronization operations are introduced during the second level of tiling. This adds an extra latency of $t_s' \times K/K'$. Hence the overhead difference between 2-level overlapped tiling and traditional tiling is:

$$\begin{aligned} \Delta Overhead' &= t_s \times (K-1) - t_s' \times K/K' - t_c \times RC'/Q \\ &\approx t_s \times (K-1) - t_s' \times \frac{K}{K'} - \frac{t_c}{Q} \times RC \times (\frac{1}{N_c} + (\frac{K'}{K})^2) \\ &= t_s \times (K - 1 - \frac{t_s'}{t_s} \times \frac{K}{K'}) - \frac{t_c}{Q} \times RC \times (\frac{1}{N_c} + (\frac{K'}{K})^2) \end{aligned}$$

As mentioned before, t_s' is much smaller than t_s (Equation 4). Thus, if K' is appropriately chosen, $\Delta Overhead'$ can be made larger than $\Delta Overhead$ as defined in 3, which shows the potential benefit of hierarchical overlapped tiling.

```
__kernel void kernel(__global float *A,
                     __global float *B) {
    int i = get_global_id(0);
    B[i] = A[i-1]+A[i]+A[i+1];
}
```

(a) OpenCL kernel code

```
cl_mem mem_A, mem_B, *p1=&mem_A, *p2=&mem_B;
...;
for(int k = 0 ; k < K; k++) {
    clSetKernelArg(kernel,0,sizeof(cl_mem),p1);
    clSetKernelArg(kernel,1,sizeof(cl_mem),p2);
    size_t global_work_size[] = {N};
    new_evt=(cl_event*)malloc(sizeof(cl_event));
    clEnqueueNDRangeKernel(queue,kernel,1,NULL,
        global_work_size,NULL,1,event,new_evt);
    event = new_event;
    swap(p1, p2);
}
```

(b) OpenCL host code

Figure 5: OpenCL code example

3. IMPLEMENTATION

3.1 OpenCL

An OpenCL program consists of two classes of components: host code and kernel code. The host code contains the control logic and usually runs on a general purpose processor, while the kernel code contains most of the computation and executes on the target device, such as an accelerator. The host code is a C/C++ program with OpenCL API invocations. The kernel code is written in OpenCL C, which is a subset of C99 with extensions. OpenCL *kernel*s resemble C procedures. During execution, the host code compiles the kernel code. This dynamic runtime compilation makes OpenCL programs portable across different devices.

OpenCL kernels follow the SPMD execution model. The host code specifies the work item organization of each kernel, where a *work item* is the unit of scheduling. Work items are grouped into *work groups*. The work groups and the work items within each work group have N dimensions ($N \leq 3$). Each work group is represented by an N-tuple work group ID (which can be accessed during execution using `get_group_id()`), and each work item also has an N-tuple work item IDs. The functions (`get_global_id()`) and (`get_local_id()`) can be used to access the global or local work item ID respectively.. The work item ID defines an N-dimensional index space. Thus, we can conceive the execution of a sequence of OpenCL kernels as a sequence of parallel loops. For example, the OpenCL code in Figure 5 is equivalent to the code in Figure 2-(a).

3.2 Implementation Overview

We implemented the hierarchical overlapped tiling optimization to evaluate its effectiveness. A diagram representing our experimental system is shown in Figure 6. The dashed arrows represent offline data flow while solid arrows represent runtime data flow. The system contains three major components: a delayed compilation mechanism, an offline analyzer and the optimizer. The offline analyzer is implemented as a pass of Cetus [12]. The optimizer is a source-to-source translator which reads the original kernel code and generates OpenCL code, which is fed to the OpenCL runtime.

Both, the offline analyzer and the optimizer use the Omega library [13] to perform the integer tuple space computations. In addition, the analyzer uses the Omega Library to generate loops from the integer tuple sets.

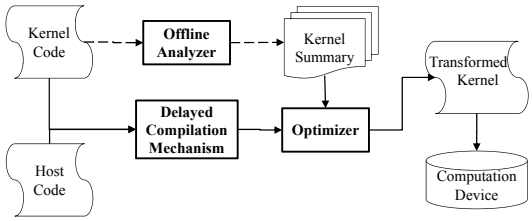


Figure 6: Framework of the automatic transformation tool.

3.3 Delayed Compilation Mechanism

In our framework, each OpenCL kernel execution (or instance) is seen as a parallel loop. This view combines information from the host code (the iteration space) and the OpenCL kernel code (the body of the loop). The framework applies overlapped tiling or hierarchical overlapped tiling to a sequence of kernel invocations. To fully automate the process, two problems must be solved: 1) The dynamic compilation unit in OpenCL is kernel, but overlapped tiling fuses across kernel boundaries. Therefore, some form of global analysis is needed. 2) The optimal value of the transformation parameters usually depend on the iteration space, which is the work size of an OpenCL kernel. However, in OpenCL the work size is specified by the host code, and sometimes the values of the work size can only be resolved after compiling the kernel code. In order to solve these two problems, we designed and implemented a delayed compilation mechanism. It postpones the compilation process (which should be done in `clBuildProgram()` in the standard OpenCL APIs) of the kernel code until a sufficient number of kernels have been created and enqueued by the host code. In order to enable the reuse of the standard compilation process, our delayed compilation mechanism is implemented as wrapper functions.

Our delayed compilation mechanism works as follows: when the host code invokes `clEnqueueNDRangeKernel()` to push a kernel instance into the command queue, the kernel instance is actually held within a `pending_kernel` object in the pending queue. Each pending kernel carries with it the work size and arguments specified by the host code. The kernel will not be compiled until:

1. a synchronization point is found during execution of the host code; this usually means that the results of the pending kernels are needed to continue execution, or
2. the number of pending pending kernels exceeds a given threshold.

Each time the compilation process is triggered, the optimizer will be invoked to select some kernels from the pending queue to apply the appropriate transformation. After compilation, the transformed kernel generated by the optimizer is executed. This way, the transformation is transparent to the programmer.

3.4 Offline Analyzer

As mentioned before, the consuming relation C_k and producing relation P_k represent the access pattern of $loop_k$. Since we view each OpenCL kernel as a parallel loop, we can also use consuming and producing relations to represent the access pattern of the kernel code (the body of the loop). Computing consuming and producing relations requires traversing the syntax tree of the kernel code to collect every global memory reference, which might be an expensive operation. If symbolic variables are allowed in the constraints of integer tuple relations, it is possible to compute the consuming and producing relations of each kernel offline to reduce the overhead of the online compilation.

The algorithm to compute consuming and producing relations makes use of symbolic range propagation [7]. This produces a conservative approximation to the range of possible values of every variable used to compute array ranges. The output of the offline analyzer is the summary for each $Kernel_k$, which contains the consuming relation C_k^A or producing relation P_k^A of every global array A accessed by the kernel body.

3.5 Optimizer

When the compilation process is triggered by the delayed compilation mechanism, the optimizer tries to apply the overlapped tiling transformation on the pending kernels. OpenCL programs can be configured for in-order execution or out-of-order execution. In in-order execution mode, kernels are executed in the same order that they are enqueued, while in out-of-order execution mode, the scheduler of OpenCL runtime is free to reorder kernels as long as the partial order specified by the programmer is respected. Fusion can be applied to a sequence of loops if the runtime is configured for in-order execution or to the topological sort of a partial order if the runtime is configured for out-of-order execution.

The optimizer uses the producing/consuming relations discussed in Section 2 to represent the dependence of kernel instances. The discussion in Section 2 only considers the situation when all the data consumed by a loop are produced by its predecessor. However, many programs do not conform to this simplification. We say that there is a *producer-consumer relation* between two kernels X and Y: if and only if kernel X is an ancestor of kernel Y (in terms of the partial execution order enforced by event objects) and the elements produced by kernel X are consumed by kernel Y. We say that X is Y's producer and Y is X's consumer. The goal is to guarantee that for each tile, the producer kernel produces all the data (array elements) that the consumer kernel needs. According to the code in Figure 5, kernel instances $Kernel_0$, $Kernel_2$, ..., $Kernel_{2n}$, ... read `mem_A` and write `mem_B`, while $Kernel_1$, $Kernel_3$, ..., $Kernel_{2n+1}$, ... read `mem_B` and write `mem_A`. Hence the producers of $Kernel_k$ include $Kernel_{k-2n} \forall n \leq k/2$. This is denoted as $Producer(k) = \{k - 2n | \forall n = 1, 2, ..., \lfloor k/2 \rfloor\}$.

We can represent producing/consuming relations in terms of data dependences. If we view the OpenCL code in Figure 5 to be equivalent to a doubly nested loop like in Figure 2-

(a), there are read-after-write dependences from iterations $(k-2n+1,i)$ and $(k-2n+1,i\pm 1)$ to iterations (k,i), $n=1,2,...,\lfloor k/2 \rfloor$. So the dependence vectors are $(2n-1,0)$ and $(2n-1,\pm 1)$, which correspond to be direction vectors $(<,=)$ and $(<,*)$. With the distance vectors or direction vectors, loop transformations can be performed as described in [3] On the other hand, the dependence vectors can be computed through producing/consuming relations of kernel instances.

Although we could use the dependence information, what we need for the transformation is only the data flow between kernels, which can be naturally represented by producing/-consuming relations. Thus, the analysis in the optimizer uses the producing/consuming relations instead of data dependences.

3.5.1 Tiling Algorithm

To apply the overlapped tiling optimization to the sequence of kernels, we first compute the symbolic tiles for each kernel, and then the actual size of the tiles is determined.

Input: Consuming and producing relations C_k and P_k for each $Kernel_k$
Output: $Tile_k$ for each $Kernel_k$
Initialization: $Tile_k = \emptyset$, for each $Kernel_k$

$Kernel_{K-1}$ = the last kernel in a topological order;
$Kernel_0$ = the first kernel in the same topological order;
$Tile_{K-1} = \{[id_0,...,id_{D-1}] : s_0 \le id_0 < s_0 + len_0 \wedge$
$... \wedge s_{D-1} \le id_{D-1} < s_{D-1} + len_{D-1}\}$;
for(each $Kernel_k$ in $Kernel_{K-1},..,Kernel_0$ in reversed topological order)
  for(each array A that is an argument read by $Kernel_k$)
    for(each $Kernel_{k'}$ which is an ancestor of $Kernel_k$)
      $Tile_{k'} = Tile_{k'} \cup P^A_{k'}(C^A_k(Tile_k))$;

Figure 7: Algorithm for symbolic tiling

Symbolic Tiling. Figure 7 shows a simplified algorithm for tiling. We assume a collection of K kernels ($kernel_0$, $kernel_1$, ..., $kernel_{K-1}$). The basic idea of this algorithm is, starting from the last kernel in topological order ($kernel_{K-1}$), traverse backwards the consumer-producer chain to determine all the data that must be computed inside each tile, so that no inter-tile communication or synchronization is needed.

$Tile_k$ is a set of *integer tuple*s where each element represents a work item ID in the resulting tile for $Kernel_k$. At the beginning, the tile for the start kernel $Tile_{K-1}$ is initialized with $\{[id_0,...,id_{D-1}] : s_0 \le id_0 < s_0 + len_0 \wedge ... \wedge s_{D-1} \le id_{D-1} < s_{D-1} + len_{D-1}\}$ in which s_i and len_i are variables whose values will be determined later. We assume without loss of generality that the work items for $Kernel_{K-1}$ are organized into D-dimensional objects. The output of this algorithm is the symbolic tile $Tile_k$ for each kernel, which contains symbol variable s_i and len_i in the constraints. The symbolic tiling algorithm can be used by the plain overlapped tiling and the hierarchical overlapped tiling, since it requires neither the work size nor the tile boundaries as input.

Determining Tile Size. With the symbolic tile for each kernel, we can estimate the memory footprint of each tile. For each global array A, a superset of the elements read (R_A) or written (W_A) within a tile can be computed as follows:

$$R_A = \bigcup_{k=0}^{K-1} C^A_k(Tile_k), \quad W_A = \bigcup_{k=0}^{K-1} P^{A^{-1}}_k(Tile_k)$$

If we only count global array elements [1], the size of memory footprint is $FP = \sum_A |R_A \cup W_A|$. FP is a function of the tile length (len), the number of kernels to fuse (K) and the work size of $Kernel_{K-1}$ (I_{K-1}). Given K and I_{K-1}, the tile size can be determined by finding a value of len that satisfies the constraint below, to guarantee that the footprint of each tile fits in cache:

$$FP = FP(len, K, I_{K-1}) < Eff\_Cache$$

Eff_Cache is the effective size of the target level of cache. For plain overlapped tiling, Eff_Cache is set to be a fraction of the shared last level cache on each processor; for the second level of hierarchical overlapped tiling, I_{K-1} should be the tile size generated by the higher level tiling and Eff_Cache is determined by the size of the private cache (L1 or L2). For other architecture such as GPUs, Eff_Cache is determined by the size of the local storage.

Determining the Loop Fusion Depth K. The discussion in the last subsection assumes that the number of kernels to fuse (K) is known. As discussed in Subsection 2.3, the amount of redundant computation introduced by plain overlapped tiling (Equation 5) and hierarchical overlapped tiling is a function of the loop fusion depth K. We provide two ways to specify the value of K: (1) the programmer specifies the value of K as an input parameter; (2) the value of K can be determined based on the performance feedback obtained using training inputs.

When using the second method for plain overlapped tiling, we define $\Delta RC(k) = RC(k) - RC(k-1)$, which is the redundant computation increment when adding $Kernel_k$ for fusion. We find that usually $\Delta RC(k)$ is a monotonically nondecreasing function, e.g., for the code in Figure 2-(a) discussed in Subsection 2.3, $\Delta RC(k) = 2 \times Q \times (k-1)$. Thus, in order to obtain the maximum $\Delta Overhead$ (Equation 3) of overlapped tiling over traditional tiling, we should stop adding new kernels when the following become true:

$$t_c \times \Delta RC(k)/Q \ge t_s \quad or \quad \Delta RC(k)/Q \ge t_s/t_c \qquad (6)$$

In the above inequality, t_s is platform dependent, while t_c is application dependent. We assume that it is possible to profile the value of t_s/t_c with a representative input data. Then, for adaptive tuning, the optimizer keeps adding new kernels from the pending queue to be transformed until inequality 6 becomes true; at that point, the number of kernels added is the desired value of K.

For hierarchical overlapped tiling, tuning must be done for

[1]Our current implementation only considers the global memory space, since the main data structures of the programs we evaluated are in global memory space. However, to be accurate, objects in the local memory space such as local memory objects and stack variables should also be taken into account.

each level of tiling to determine K and K' separately, with different average cost of synchronization (t_s or t'_s).

Create Thread-local Buffers. For some architectures such as GPGPUs, different OpenCL address spaces `__global`, `__local` and `__private` are mapped to physical storages with different speeds. Thus, it is beneficial for performance to create thread-local buffers and promote working data into faster storages. For multicore platforms with transparent caches, the different OpenCL address spaces are all mapped to main memory and therefore the use of these address spaces does not impact performance. However, since different tiles compute redundantly some of the array elements, the data for each tile must be privatized for correctness.

For each work item we create a local buffer for each array A of size $|R_A \cup W_A|$, so that the thread-local buffers are large enough to hold every element of the array base A accessed by each tile. In our implementation, we only create buffers of regular shape, e.g., for 2D data we always create a rectangular area for the thread local buffer which covers every element accessed by the work item. This strategy simplifies the loop boundary control and array element index translation of the generated code, but can introduce more redundant computation when the dependences are irregular.

3.6 Other Implementation Issues

In order to avoid recompilation by our delayed compilation mechanism, we implemented a cache to hold previous compilation results. Each compilation result (compiled code of transformed kernel) is indexed by the sequence of kernel names, and the value of the tiling-related arguments of each kernel. An argument is tiling-related only if it is involved in any constraints of the consuming or producing relation of the kernel of which it is an argument. For instance, the argument `A` in Figure 5 is not tiling-related. Thanks to the compilation cache, if the same sequence of kernels occurs more than once during execution, previous compilation results can be accessed to avoid recompilation. For OpenCL programs with stable repetition units, such as stencil code, the total number of compilation passes can be reduced to 1 or 2: one for the steady state, the other for the epilog if it exists. If the OpenCL program does not have a stable repetition pattern, the total compilation time could increase. But if the OpenCL programs run on a heterogeneous platform with multiple computation devices, it is possible to overlap the transformation and compilation with the kernel execution.

For stencil programs which iteratively execute the same kernel, delayed compilation has an effect similar to loop unrolling, which may results in code explosion and hurt instruction locality (and hence dramatically increase instruction cache misses) when the fusion depth is large. To avoid code explosion, our implementation includes a pattern matching-based "re-rolling" pass where the generated code only contains two loops: an outer loop, whose iteration count is the number of loops being fused, and an inner loop, with a variable number of iterations that depends on the value of the induction variable of the outer loop.

4. EVALUATION ENVIRONMENT

4.1 Target platform

We evaluate the efficiency of the proposed transformation on an SMP workstation with 4 Intel Xeon L7555 processors running at 1.87GHz. Each processor has 8 cores, sharing a 24MB unified L3 cache on chip. Each core contains a 256KB private L2 cache and 32KB L1 D-cache. SMT is disabled for each core, so there is one hardware thread per core. After transformation, the OpenCL code is compiled using an experimental OpenCL compiler from Intel Labs.

4.2 Benchmarks

For the evaluation we use 8 benchmarks: 1D/2D/3D-Jacobi, PathFinder, Poisson, Biharmonic, HotSpot and Cell. The first 3 benchmarks (1D/2D/3D-Jacobi) are Jacobi iterations for synthetic linear systems; PathFinder uses dynamic programming to find a minimum weighted path; Poisson is a numerical solver of the poisson equation, calculating the Laplace operator [5] over a 2D grid with the 5-point stencil. Biharmonic is the numerical PDE solver calculating the Biharmonic operator [5] over a 2D grid with a 13-point stencil. HotSpot implements a chip temperature estimation model[11]. Cell [4] is a 3D game of life. For each application, the operation in the body of the stencil loop is implemented as an OpenCL kernel. The inputs of the benchmarks are listed in Table 1.

	Data Dimension	Problem Size	Points of Stencil
1D-Jacobi	1	64K	3
2D-Jacobi	2	256x256	9
3D-Jacobi	3	64x64x64	27
PathFinder	1	100K	3
Poisson	2	256x256	5
Biharmonic	2	256x256	13
HotSpot	2	512x512	9
Cell	3	60x60x60	27

Table 1: Benchmarks

For some stencil programs, such as Jacobi, the number of steps of the outer loop depends on a convergence test. This requires a synchronization between kernel code and host code that prevents loop fusion. To enable the overlapped and hierarchical overlapped tiling optimizations we modified the code so that the convergence test (and as result the synchronization) only occurs every 1024 iterations. The total iterations number for the benchmarks without convergence test is 16,384.

We use `pthread_setaffinity_np()` to set the appropriate affinity for each worker thread to guarantee that the threads executing the sub-tiles within the same tile communicate through a shared cache for hierarchical overlapped tiling.

5. EXPERIMENTAL EVALUATION

In this Section, we present our experimental results. Section 5.1 presents the main results, Section 5.2 discusses parameter sensitivity, and Section 5.3 discusses compilation overhead.

5.1 Performance Overview

Figure 8 shows the performance speedup of traditional tiling, overlapped tiling and hierarchical overlapped tiling relative to the original OpenCL code. Since OpenCL only supports 2 levels of work item organization, a 2-level hierarchical

overlapped tiling is used for each benchmark. For overlapped tiling and hierarchical overlapped tiling, the figure shows the speedup for the best value of the loop fusion depth K, as shown in Table 2 (and discussed in the next Section). In general, the performance of hierarchical overlapped tiling is always better than that of plain overlapped tiling, and overlapped tiling is always better than that of traditional tiling, with the exception of Cell and 3D-Jacobi. For Cell and 3D-Jacobi, the performance curves of the three tiling transformations are very similar. The reason is that their main data structure is 3-dimensional and the amount of redundant computation grows quartically with the fusion depth K. Therefore, there are not many opportunities for overlapped tiling and hierarchical overlapped tiling. In our experiments, overlapped tiling and the 2-level hierarchical overlapped tiling achieves an average speedup of 18% and 37% over traditional tiling, respectively.

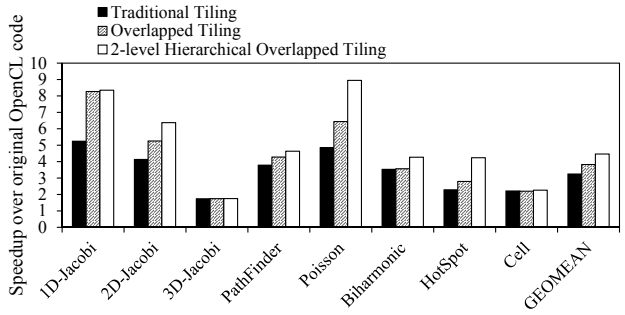


Figure 8: Speedup of traditional tiling, overlapped tiling and hierarchical overlapped tiling over the original openCL code.

5.2 Parameter Sensitivity

Loop Fusion Depth for the First Level of Tiling. Figure 9 shows the performance of overlapped and hierarchical overlapped tiling as the value of the loop fusion depth K changes. For overlapped tiling, there is only one value of K; for 2-level hierarchical overlapped tiling, K is the loop fusion depth for the first level of tiling, while K' is the value of loop fusion depth for the second level of tiling (K' is kept constant at 2 for the experiments in Figure 9). As discussed in Subsection 2.3 and 2.4, K, determines the amount of redundant computation introduced by overlapped and hierarchical overlapped tiling, and thus the overall speedup over traditional tiling.

In Figure 9, lines a, b and c show the speedup of traditional tiling, overlapped tiling, and 2-level hierarchical overlapped tiling over the original OpenCL code, respectively. In addition, we manually modified the overlapped tiling code to remove the redundant computation (hence, there are race conditions and the results of the executed code are not guaranteed to be correct), and its performance is shown by Line d. OpenCL standard does not support a global barrier across work item groups, which is required for traditional tiling (See Figure 1-(b)); thus, the barriers used for traditional tiling (Line a in Figure 9) are barriers that we implemented with low level primitives. However, overlapped tiling and 2-level hierarchical overlapped tiling only require synchro-

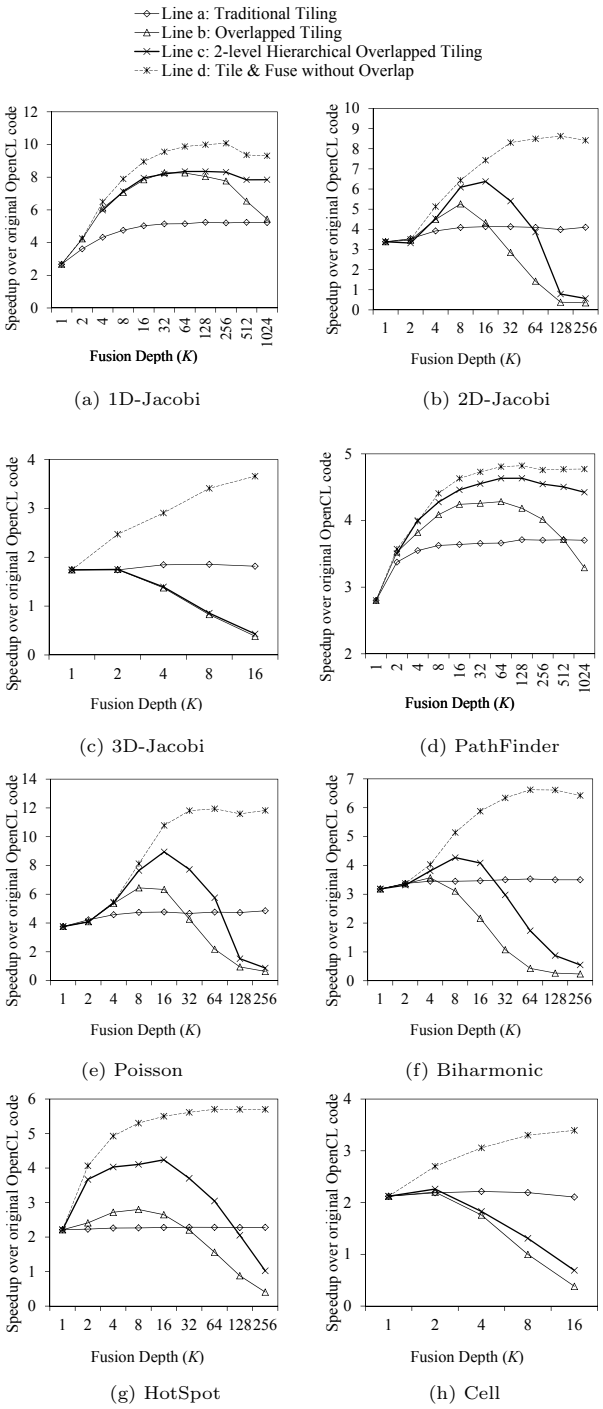


Figure 9: Evaluating the performance overlapped tiling and hierarchical overlapped tiling by scaling fusion depth K.

nization within the work item groups, which is supported by the OpenCL standard. The discrepancy between overlapped tiling and Line d shows the overhead of the redundant computation introduced by overlapped tiling; the difference between traditional tiling and Line d shows the synchronization overhead saved by fusing the kernels. In Figure 9 we can see that Line d typically grows as the loop fusion depth

K increases. This is because the number of syncrhonization operations removed grows with the depth. If we do not count the cost of the redundant computation introduced, the performance benefit is always positive (if $RC = 0$, $\Delta Overhead$ in Equation 3 always increases when K increases). In fact, Line d defines the upper bound of performance for overlapped and hierarchical overlapped tiling.

For the two 1D benchmarks (1D-Jacobi and PathFinder), both plain overlapped tiling and hierarchical overlapped tiling can achieve significant speedup over traditional tiling, because the growth rate of redundant computation for overlapped tiling is low. For the 2D benchmarks (2D-Jacobi, Poisson, Biharmonic and HotSpot) the growth rate of the redundant computation is higher than for the 1D benchmarks. Thus, the interval of benefit (the values of fusion depth K for which lines b or c are above line a) of the 2D benchmarks is smaller than that of the 1D benchmarks for both, overlapped and hierarchical overlapped tiling. The figure also shows that the interval of benefit of hierarchical overlapped tiling is always bigger than that of plain overlapped tiling. Within the 2D benchmarks, Biharmonic shows the least speedup with overlapped tiling over traditional tiling. This is because the stencil of Biharmonic depends on 13 neighboring points versus 9 for 2D-Jacobi, 5 for Poisson, and 5 for Hotspot (see Table 1). As a result, the redundant computation of Biharmonic grows faster than that of the other 2D benchmarks. However, hierarchical overlapped tiling still achieves speedup over traditional tiling. Since the input data of Cell and 3D-Jacobi are 3-dimensional, the amount of redundant computation increases so fast that there is no opportunity for overlapped tiling.

When comparing hierarchical overlapped tiling versus overlapped tiling the plots in Figure 9 show that hierarchical overlapped tiling performs better as the value of loop fusion depth K increases (the only exception occurs with the 3D benchmarks where all 3 tiling mechanisms behave the same), because the growth rate of redundant computation is lower for hierarchical overlapped tiling than for plain overlapped tiling. Hierarchical overlapped tiling is less sensitive to the value of K than overlapped tiling because, as mentioned above, the beneficial region of hierarchical tiling is larger than that of overlapped tiling.

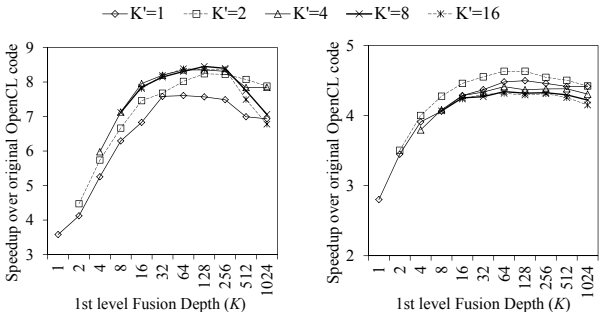
The adaptive fusion mechanism described in Subsection 3.5 uses inequality 6 and the value of t_s/t_c profiled with a smaller input set to determine the choice of the loop fusion depth K value. Our results show that the optimal values for K found using the adaptive mechanism are the same as the ones found using the empirical search in Figure 9, and shown in Table 2.

Loop Fusion Depth for the Second Level of Tiling. Compared to the loop fusion depth K for the first level tiling of hierarchical overlapped tiling, the tuning of loop fusion depth K' for the second level tiling is more involved. This is partially because the performance of the second level tiling depends on the first level of tiling. Figure 10 shows the performance of hierarchical overlapped tiling for 1D-Jacobi and PathFinder with different pairs of K and K'. We can see that $K' = 2$ is the best choice for PathFinder; but there is no obvious optimal value for 1D-Jacobi. Thus, manual tuning

	Overlapped Tiling	Hierarchical Overlapped Tiling
1D-Jacobi	$K = 32$	$K = 64$
2D-Jacobi	$K = 8$	$K = 16$
3D-Jacobi	$K = 2$	$K = 2$
PathFinder	$K = 32$	$K = 64$
Poisson	$K = 8$	$K = 16$
Biharmonic	$K = 4$	$K = 8$
HotSpot	$K = 4$	$K = 16$
Cell	$K = 1$	$K = 2$

Table 2: Loop Fusion depth K determined by the adaptive mechanism.

may be required to find the optimal fusion depth K' for the second level of hierarchical overlapped tiling. However, the performance impact of the second level tiling is significantly smaller than that of the first level tiling. For instance, when K' is set to 2, the average performance loss is less than 5% compared to the best K'.



(a) 1D-Jacobi (b) PathFinder

Figure 10: Fusion depth K' for the second level of hierarchical overlapped tiling.

Input Size. Figure 11 shows the speedup of hierarchical overlapped tiling over plain overlapped tiling for the different benchmarks, as the input size increases. The speedups are computed using the best value of K. The figure shows that hierarchical overlapped tiling performs better than overlapped tiling for the 2D-benchmarks. It also shows, that the performance difference between hierarchical overlapped tiling and plain overlapped tiling becomes smaller as the input data size increases. This is because since the total number of worker threads remains constant, the tile size is determined by the input data size. With smaller tiles, the redundant computation has a higher impact; thus, overlapped tiling is less efficient, while hierarchical overlapped has more opportunities to reduce the overheads introduced by the redundant computation. With larger tiles, the redundant computation has less impact, and as a result, there is less difference between overlapped and hierarchical overlapped tiling. For the 3D benchmarks 3D-Jacobi and Cell, since the amount of redundant computation grows so fast, the optimal K for both plain overlapped tiling and hierarchical overlapped tiling is less than 2, so there is no obvious performance discrepancies between the two schemes.

5.3 Compilation Overhead

Since our tool transforms OpenCL kernel code at runtime, the overheads introduced need to be considered. The over-

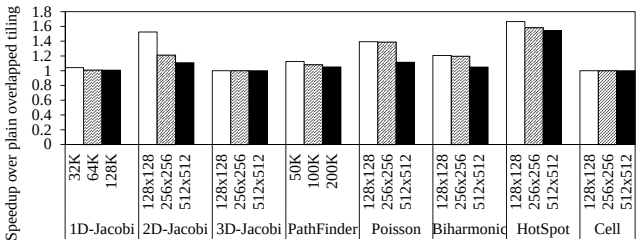


Figure 11: Speedup of hierarchical overlapped tiling over plain overlapped tiling with different input sizes. The horizontal axis is the input size for each benchmark.

head consists of two parts: The OpenCL runtime that transforms the kernel code and the execution of the Omega Library. The compilation cache described in Subsection 3.6 can help reduce both parts of the runtime overhead: if the sequence of kernels selected for optimization are the same as a previous sequence, no compilation needs to be done. For the benchmarks used in this paper, the OpenCL runtime compilation needs to be invoked at most twice, one for the steady state and another for the epilog. Figure 12 shows the compilation times of overlapped tiling and hierarchical overlapped tiling, normalized to the compilation time of the original program. On average, overlapped tiling requires 37% more compilation time, while hierarchical overlapped tiling costs about 60% more.

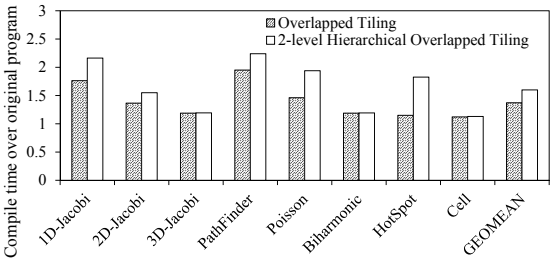


Figure 12: Compile time for overlapped tiling and hierarchical overlapped tiling, normalized to the compile time of the original OpenCL code.

6. RELATED WORK

Loop tiling is a traditional but effective optimization for performance. Numerous optimizations based on the tiling of iteration spaces have been proposed for improving data locality [27, 28, 1, 2, 24, 9, 22], or exploiting parallelism [26, 29, 6, 10, 30].

The closest work to our overlapped tiling is that of Krishnamoorthy et al.[16]. Their approach uses the polyhedral model of computation and manipulates regular data dependencies. Other works such as Ripeanu et al. [23] and Meng et al. [21] describe performance models to predict the optimal amount of redundant computation for stencil computation in a grid environment with message passing or for GPUs, respectively. However, no fully automated tool for overlapped tiling is discussed in these papers. Our work uses producer/consumer relations to represent dependence of kernel instances, and the transformation tool designed and implemented in this paper is fully automated and transparent to OpenCL programs. The most important difference between our work and existing work is the hierarchical overlapped tiling transformation proposed in this paper. The overhead of redundant computation is the main drawback of the overlapped tiling approach; by applying overlapped tiling hierarchically, we can decrease this overhead.

Bondhugula et al. design and implement Pluto [8], which can automatically transform loops for parallelism and locality based on the polyhedral model. Their transformation techniques includes only traditional tiling; overlaps between tiles are not considered. Tang et al. implemented the Pochoir compiler [25], which automates a trapezoidal decomposition for stencil code. However, their decomposition algorithm does not consider overlap, either.

Other publications that discuss how to take advantage of the hierarchy of the hardware include Liu et al. [19], that proposed a cache hierarchy-aware tile scheduling technique to maximize data reuse; and Leung et al. [18] that implement a C-to-CUDA compiler which performs hierarchical decomposition for multiple GPUs. The techniques presented in these papers are orthogonal to our proposed transformation. In fact, their techniques and ours could be combined.

7. CONCLUSION

In this paper, we propose a new transformation, hierarchical overlapped tiling. By creating hierarchical overlapping tiles, we reduce communication overhead among tiles while introducing smaller amount of redundant computation compared to plain overlapped tiling. We implemented the proposed transformations for OpenCL programs. Experimental results show that on average overlapped tiling and hierarchical overlapped tiling achieves 18% and 37% speedup over traditional tiling, respectively.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Awards CNS 1111407 and CCF 0702260, and by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

9. REFERENCES

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *Computers, IEEE Transactions on*, C-30(5):341 –356, may 1981.

[2] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing, ACM/IEEE 2000 Conference*, page 31, nov. 2000.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, 2001.

[4] M. Alpert. Not just Fun and Games. *Scientific American*, 4, 1999.

[5] W. F. Ames. *Numerical Methods for Partial Differential Equations.* Academic, San Diego, CA, sencond edition, 1977.

[6] 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, pages 153–162, New York, NY, USA, 2001. ACM.

[7] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings. of 9th International Parallel Processing Symposium, 1995.*, pages 357 –363, apr 1995.

[8] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[9] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 279–290, New York, NY, USA, 1995. ACM.

[10] K. Högstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '99, pages 201–211, New York, NY, USA, 1999. ACM.

[11] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusam. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 878–883, New York, NY, USA, 2004. ACM.

[12] T. Johnson, S.-I. Lee, L. Fei, A. Basumallik, G. Upadhyaya, R. Eigenmann, and S. Midkiff. Experiences in using cetus for source-to-source transformations. In *Languages and Compilers for High Performance Computing*, volume 3602 of *Lecture Notes in Computer Science*, pages 922–922. 2005.

[13] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library interface guide. Technical report, 1995.

[14] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[15] G. Kreisel and J.-L. Krivine. *Elements of mathematical logic*. North-Holland Pub. Co., 1967.

[16] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. 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.

[17] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[18] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 51–61, New York, NY, USA, 2010. ACM.

[19] J. Liu, Y. Zhang, W. Ding, and M. Kandemir. On-chip cache hierarchy-aware tile scheduling for multicore machines. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 161 –170, april 2011.

[20] D. B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24:121–145, 1977.

[21] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[22] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 111–120, New York, NY, USA, 1991. ACM.

[23] M. Ripeanu, A. Iamnitchi, and I. Foster. Cactus application: Performance predictions in grid environments. In *Euro-Par 2001 Parallel Processing*, volume 2150 of *Lecture Notes in Computer Science*, pages 807–816, 2001.

[24] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 215–228, New York, NY, USA, 1999. ACM.

[25] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[26] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452 –471, oct 1991.

[27] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[28] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[29] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM.

[30] D. Wonnacott. Time skewing for parallel computers. In *Languages and Compilers for Parallel Computing*, volume 1863, pages 477–480, 2000.