Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs

Peng Di, Ding Ye, Yu Su, Yulei Sui and Jingling Xue
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia

***Abstract*—Automatically parallelizing loop nests into CUDA kernels must exploit the full potential of GPUs to obtain high performance. One state-of-the-art approach makes use of the polyhedral model to extract parallelism from a loop nest by applying a sequence of affine transformations to the loop nest. However, how to automate this process to exploit both intra- and inter-SM parallelism for GPUs remains a challenging problem. Presently, compilers may generate code significantly slower than hand-optimized code for certain applications.**

This paper describes a compiler framework for tiling and parallelizing loop nests with uniform dependences into CUDA code. We aim to improve two levels of wavefront parallelism. We find tiling hyperplanes by embedding parallelism-enhancing constraints in the polyhedral model to maximize intra-tile, i.e., intra-SM parallelism. This improves the load balance among the SPs in an SM executing a wavefront of loop iterations within a tile. We eliminate parallelism-hindering false dependences to maximize inter-tile, i.e., inter-SM parallelism. This improves the load balance among the SMs executing a wavefront of tiles. Our approach has been implemented in PLUTO and validated using eight benchmarks on two different NVIDIA GPUs (C1060 and C2050). Compared to PLUTO, our approach achieves 2 – 5.5X speedups across the benchmarks. Compared to highly hand-optimized 1-D Jacobi (3 points), 2-D Jacobi (5 points), 3-D Jacobi (7 points) and 3-D Jacobi (27 points), our speedups, 1.17X, 1.41X, 0.97X and 0.87X with an average of 1.10X on C1060 and 1.24X, 1.20X, 0.86X and 0.95X with an average of 1.06X on C2050, are competitive.

***Keywords*-GPU; Loop Tiling; Loop Parallelization**

I. Introduction

Parallelization of loop nests continues to drive much of the ongoing research in parallel computing. Loop nests with uniform dependences are widely used in scientific and engineering applications, including image processing, computational electromagnetics and numerical analysis [10], [26], [31], [36], [37], [38]. Their efficient implementation on GPUs is becoming increasingly important as GPUs have recently emerged as powerful fine-grained parallel co-processors for general-purpose computing.

Traditionally, the concept of uniform dependence applies only to a set of perfectly nested loops. In this paper, we consider both perfectly and imperfectly nested loops with uniform dependences. Given a loop nest (normalized with all loops in the same depth being associated with the same loop variable), a dependence between two references is *uniform* if their dependence distance is constant. Jacobi and SOR, given in Figure 1, are imperfectly and perfectly nested loops with uniform dependences, respectively.

```
for (t=1;t<=T;t++){ //DOSEQ
  for (i=1;i<=I;i++) //DOALL
S0:  B[i]=(A[i-1]+A[i]+A[i+1])/3;
  for (i=1;i<=I;i++) //DOALL
S1:  A[i]=B[i];
}
```

(a) Jacobi: imperfectly nested loops

```
for (t=1;t<=T;t++){ //DOSEQ
  for (i=1;i<=I;i++) //DOACROSS
S:   A[i]=(A[i-1]+A[i]+A[i+1])/3;
}
```

(b) SOR: perfectly nested loops

Figure 1: Loop nests with uniform dependences.

In this paper, we restrict ourselves to stencil computations. There are several studies on parallelizing stencil computations on GPUs by hand [8], [22], [25], [37]. However, manual development of high-performance CUDA kernels can be time-consuming and error-prone. Among several alternative approaches, automatic parallelization is promising yet challenging. Recently, based on the polyhedral model, attempts have been made to implement automatic C-to-CUDA parallelizing compilers for GPUs [2], [4], [28].

An important open problem faced by the polyhedral model is how to synthesize a sequence of affine transformations from a huge space of valid solutions. To execute a tiled loop nest efficiently on a cluster of CPU nodes, one well-known communication-minimal approach is to find appropriate tiling hyperplanes so that the inter-tile communication is miminized [6], [40]. Although a significant body of prior work focuses on optimizing hyperplane generation strategies and tiling heuristics [1], [13], [14], [17], [21], [29], [30], these existing solutions, once directly deployed for GPUs, lead to poor performance for some applications.

In GPUs, the degree of parallelism across the *Streaming Multiprocessors (SMs)* and among the *Streaming Processors (SPs)* in an SM is the key to improving performance [3],

[15], [34]. To parallelize a tiled loop nest on GPUs, the wavefronts within a tile are pipelined for parallel execution to exploit intra-SM parallelism and the wavefronts across the tiles are pipelined for parallel execution to exploit inter-SM parallelism. By applying directly the CPUs-oriented communication-minimal approach [6], [40], [41] to GPUs, severe load imbalance may occur due to pipeline fill-up and drain delay. In this work, we overcome this limitation by employing two new parallelism-exposing transformations for GPUs. This paper makes the following contributions:

- We introduce an affine tiling framework that performs automatic C-to-CUDA parallelization from loop nests with uniform dependences. Our framework aims to maximize two levels of wavefront parallelism by reducing pipeline fill-up and drain delay: intra-tile, i.e., intra-SM parallelism by finding appropriate tiling hyperplanes and inter-tile, i.e., inter-SM parallelism by eliminating parallelism-hindering false dependences.
- We have compared our framework with PLUTO, an existing C-to-CUDA compiler [4], using eight stencil kernels and some hand-tuned code on two NVIDIA GPUs, C1060 and C2050. Compared to PLUTO, our speedups are 2 – 5.5X for these benchmarks. Compared to highly hand-optimized 1-D Jacobi (3 points), 2-D Jacobi (5 points), 3-D Jacobi (7 points) and 3-D Jacobi (27 points), our speedups, 1.17X, 1.41X, 0.97X and 0.87X with an average of 1.10X on C1060 and 1.24X, 1.20X, 0.86X and 0.95X with an average of 1.06X on C2050, are considered to be competitive.

Our preliminary results demonstrate that the two optimizations proposed for GPUs are quite promising.

II. Background

A. GPU Architecture and CUDA Programming Model

1) Two-Level Parallelism: The GPU architecture is based on a scalable array of SMs. Each SM comprises a number of SPs. An execution of a CUDA kernel launches a set of thread blocks and each thread block can contain hundreds of threads, with every 32 threads grouped into a warp. SPs on one SM execute a warp of 32 threads at a time; and all threads in one block are not able to be executed on two or more SMs at a time. Since threads from different blocks can be executed on SMs concurrently, there exist two-levels of fine-grained parallelism for a running kernel: inter-SM and intra-SM (i.e., among the SPs in the same SM).

2) Bank Conflicts: Bank conflicts occur when the threads in one warp access different shared memory addresses in the same bank. The threads with bank conflicts are forced to access shared memory sequentially. According to [27], if the threads in one warp access a sequence of continuous shared memory addresses (i.e., exhibit stride-1 assesses), there will be no bank conflicts. If the accesses are made in stride-2, for example, two-way bank conflicts will occur. Then a pair of threads must each wait for the other to finish.

```
for (t=1;t<=T;t++)
    for (i=1;i<=I;i++)
S:       A[i]=0.5*(A[i]+A[i+1]);
```

Figure 2: A motivating example.

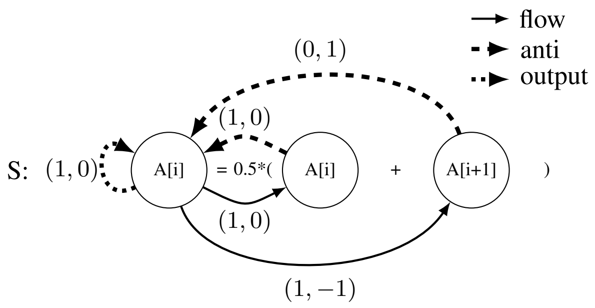


Figure 3: DDG for the loop nest given in Figure 2 with each dependence labelled by its distance vector.

B. Loop Transformations

1) Polyhedral Representation: An m-D loop nest can be represented by a polyhedron, which is a set of points surrounded by finitely many hyperplanes and can be described by a set of affine inequalities. For example, the loop nest in Figure 2 can be statement-wisely represented as follows:

$$\mathcal{R}_s \begin{pmatrix} \vec{i}_S \\ \vec{g} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} t \\ i \\ T \\ I \\ 1 \end{pmatrix} \geq 0 \qquad (1)$$

where $\mathcal{R}_S$ is an affine matrix representing the loop bound constraints, $\vec{i}_S$ is the iteration vector, and $\vec{g}$ is a vector of global parameters (which are usually the loop bounds).

2) Dependence: Data dependence analysis for arrays determines when two array references refer to the same element. The *data dependence graph* (DDG) $G = (V, E)$ is a directed multi-graph with each vertex representing a statement S and an edge $e_{S_s,S_t} \in E$ from S_s to S_t indicating a dependence between the source and target conflicting accesses in S_s and S_t, respectively. $\mathcal{D}_S$ is the *iteration space* of statement S. If $\vec{i}_s \in \mathcal{D}_{S_s}$ and $\vec{i}_t \in \mathcal{D}_{S_t}$ are dependent through edge $e_{S_s,S_t} \in E$, we write $\langle \vec{i}_s, \vec{i}_t \rangle \in \mathcal{P}_{e_{S_s,S_t}}$, where $\mathcal{P}_{e_{S_s,S_t}}$ is the *dependence polyhedron* of e_{S_s,S_t}. In the important special case when $\vec{i}_t - \vec{i}_s$ is a constant $\vec{d}$ for all $\langle \vec{i}_s, \vec{i}_t \rangle \in \mathcal{P}_{e_{S_s,S_t}}$, then $\vec{d} = \vec{i}_t - \vec{i}_s$ is known as a *distance vector*. A loop nest is said to have *uniform dependences* if all its dependences can be expressed as distance vectors.

The loop nest given in Figure 2 has only uniform dependences. Table I lists its five dependences and their dependence polyhedra. As is customary, flow (i.e., RAW) dependences are true dependences and anti (i.e., WAR) and output (i.e., WAW) dependences are false dependences. The DDG for this example is shown in Figure 3.

3) Affine Transformations: A 1-D affine transformation for statement S is an affine function defined by:

$$\phi_S(\vec{i}_S) = (c_1, \ldots, c_m)\vec{i}_S + \alpha_S \quad (2)$$

where $(c_1, \ldots, c_m), \vec{i}_S \in \mathbb{Z}^m$ and $\alpha_S \in \mathbb{Z}$. ϕ_S can be interpreted as a partitioning hyperplane with its normal being $(c_1, \ldots, c_m)$, which maps each instance of statement S to a new instance in the 1-D transformed iteration space.

An m-D affine tiling transformation Φ_S is represented as a set of m linearly independent 1-D affine functions. If ϕ_S^k represents the k-th affine function (tiling hyperplanes), then $\Phi_S = (\phi_S^1, \ldots, \phi_S^m)$ maps each instance of statement S into a new instance in the m-D transformed iteration space.

III. Two-level Parallelism Transformations

In this section, we introduce our framework to tiling and parallelizing loop nests with uniform dependences to exploit both intra- and inter-SM parallelism on GPUs. Our example is given in Figure 2. In Section III-A, we review the communication-minimal tiling transformations built for a cluster of CPU nodes [6], [40], [41] and see how they may lead to poor performance on GPUs due to long pipeline fill-up and drain delay. We overcome such load imbalance in two ways. In Section III-B, we improve intra-tile parallelism by finding better tiling hyperplanes in the polyhedral model. In Section III-C, we improve inter-tile parallelism by eliminating parallelism-hindering false dependences.

A. Communication-Minimal Transformations

To preserve the dependences in a given loop nest, a legal tiling must satisfy all its dependences [5], [40], [41].

Theorem 1 (Legality of Tiling)**.** *Consider a loop nest with $G = (V, E)$ as its DDG. $\Phi_{S_1}, \Phi_{S_2}, \ldots$ are legal (statement-wise) tiling hyperplanes iff $\forall e_{S_s,S_t} \in E$, $\forall k \in [1, m]$:*

$$\phi_{S_t}^k(\vec{i}_t) - \phi_{S_s}^k(\vec{i}_s) \geq 0, \langle \vec{i}_s, \vec{i}_t \rangle \in \mathcal{P}_{e_{S_s,S_t}} \quad (3)$$

Consider the example in Figure 2 with the five dependences listed in Table I. To construct its tiling hyperplanes $\Phi_S \in \mathbb{Z}^{2\times 2}$, we apply this theorem to the example. There are many valid solutions with different degrees of parallelism, which will have varying degrees of impact on performance. For the purposes of minimizing the inter-tile communication, $\Phi_S = \left[\begin{smallmatrix}1 & 0\\ 1 & 1\end{smallmatrix}\right]\left[\begin{smallmatrix}t\\ i\end{smallmatrix}\right] + \left[\begin{smallmatrix}0\\ 0\end{smallmatrix}\right]$, which is illustrated in Figure 4, is recommended [6], [40], [41]. In the figure, different instances of statement S are represented by dots. In the larger tile on the right, the solid/dashed/dotted arrows denoting flow/anti/output dependences between loop iterations. In the center, the three dependences between tiles are also depicted (without distinguishing flow from false dependences).

Given this tiling transformation, we can map tiles to thread blocks and loop iterations in a tile to the threads in a block by exploiting two levels of wavefront parallelism. In the CUDA code given in Figure 5, the first two loops enumerate all the tiles and the inner two the loop iterations within a tile.

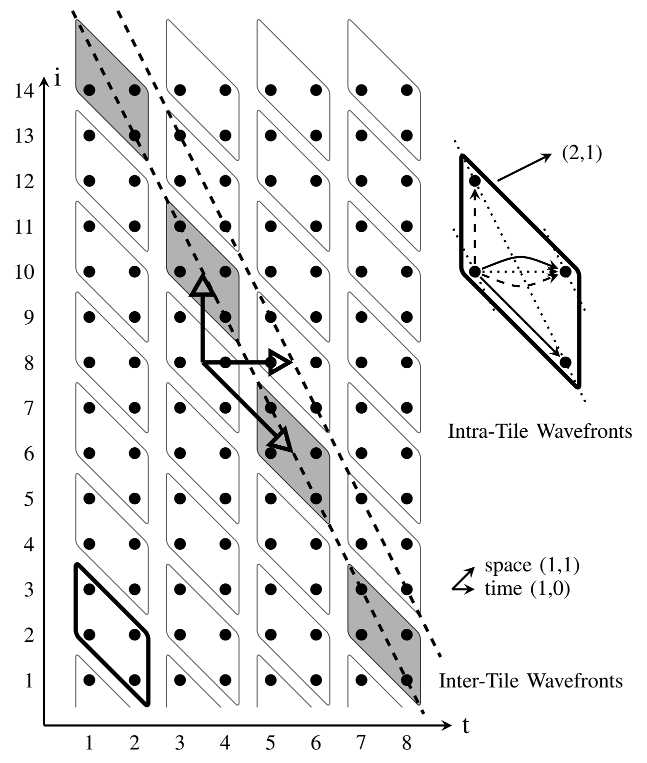


Figure 4: Iteration space tiling of the loop nest in Figure 2 with $\Phi_S = \left[\begin{smallmatrix}1 & 0\\ 1 & 1\end{smallmatrix}\right]\left[\begin{smallmatrix}t\\ i\end{smallmatrix}\right] + \left[\begin{smallmatrix}0\\ 0\end{smallmatrix}\right]$. The tile size used is 2×2.

To satisfy the three inter-tile data dependences, the tiles are grouped into wavefronts, which are pipelined for parallel execution along the direction $(1, 1)$ in the iteration space representing the tiles. One wavefront is highlighted with its tiles depicted in gray. To exploit inter-tile parallelism, different wavefronts are executed sequentially in a pipelined manner while the tiles in the same wavefront can be executed in parallel on different SMs. Therefore, the SMs are synchronized every time after a wavefront has been executed.

Like inter-tile wavefronts, the loop iterations in a tile are also scheduled to exploit intra-tile wavefront parallelism, as illustrated in the right part of Figure 4. To satisfy the five intra-tile dependences, the wavefronts in a tile are pipelined along $(2, 1)$ and the loop iterations in the same wavefront can be executed in parallel in different threads on the SPs (in an SM). Due to coalesced loads and stores used, these threads are synchronized at the end of each wavefront.

However, minimizing inter-tile communication alone often leads to long pipeline fill-up and drain delay for both intra- and inter-tile wavefronts, as can be visualized in Figure 4. The resulting load imbalance, which may be insignificant for a cluster of CPUs, can be problematic

Dependence	Type	Source $\vec{i_s}$ (t_s, i_s)	Target $\vec{i_t}$ (t_t, i_t)	Distance vector $\vec{d}$	Dependence Polyhedron
e_1: from A[i] (LHS) to A[i] (RHS)	flow	(t, i)	$(t+1, i)$	$(1, 0)$	$\mathcal{P}_{e_1}$: $t_s = t_t - 1$, $i_s = i_t$, $2 \le t_t \le T$, $1 \le i_t \le I$
e_2: from A[i] (LHS) to A[i+1] (RHS)	flow	(t, i)	$(t+1, i-1)$	$(1, -1)$	$\mathcal{P}_{e_2}$: $t_s = t_t - 1$, $i_s = i_t + 1$, $2 \le t_t \le T$, $1 \le i_t \le I-1$
e_3: from A[i+1] (RHS) to A[i] (LHS)	anti	(t, i)	$(t, i+1)$	$(0, 1)$	$\mathcal{P}_{e_3}$: $t_s = t_t$, $i_s + 1 = i_t$, $1 \le t_t \le T$, $1 \le i_t \le I-1$
e_4: from A[i] (RHS) to A[i] (LHS)	anti	(t, i)	$(t+1, i)$	$(1, 0)$	$\mathcal{P}_{e4}$: $t_s + 1 = t_t$, $i_s = i_t$, $1 \le t_t \le T-1$, $1 \le i_t \le I$
e_5: from A[i] (LHS) to A[i] (LHS)	output	(t, i)	$(t+1, i)$	$(1, 0)$	$\mathcal{P}_{e_5}$: $t_s + 1 = t_t$, $i_s = i_t$, $1 \le t_t \le T-1$, $1 \le i_t \le I$

Table I: Dependences in the loop nest given in Figure 2.

```
/* Inter-tile loops */
for(tt=0;tt<=floor(2*T+I,256);tt++){ //DOSEQ
  for(ii=max(ceil(tt,2),ceil(256*tt-T,256))+blockIdx.x;
      ii<=min(floor(T+I,256),floor(256*tt+I+255,512),tt);ii+=gridDim.x){//DOALL
  /* Intra-tile loops */
    /* Code for coalesced loads from shared memory */
    for(t'=max(3,256*tt,256*ii+1,512*ii-I,512*tt-512*ii+1);
        t'<=min(2*T+I,256*tt+510,512*ii+509,256*ii+T+255,512*tt-512*ii+I+510);t'++){ //DOSEQ
      for(i'=max(ceil(t'+1,2),256*ii,t'-T,-256*tt+256*ii+t'-255)+threadIdx.x;
        i'<=min(floor(t'+I,2),256*ii+255,t'-1,-256*tt+256*ii+t');i'+=blockDim.x)//DOALL
S:        A[-t'+2*i']=0.5*(A[-t'+2*i']+A[-t'+2*i'+1]);
      __syncthreads();
    }
    /* Code for coalesced stores into shared memory */
  }
  /* Code for synchronizing blocks */
}
```

Figure 5: CUDA kernel generated for the code in Figure 2 using the communication-minimal tiling hyperplanes in Figure 4.

for GPUs that rely on massive fine-grained parallelism to achieve high performance. To alleviate this problem, Section III-B focuses on improving intra-tile parallelism while Section III-C focuses on improving inter-tile parallelism.

B. Improving Intra-Tile Parallelism

We improve intra-tile parallelism by embedding parallelism-enhancing constraints in the polyhedral model so that better tiling hyperplanes can be found.

Definition 1 (Balanced Intra-Tile Wavefronts)**.** The wavefronts for a statement S in a tile are *balanced* if they are pipelined along the normal of a tiling hyperplane in Φ_S.

In practice, this implies that statement S will be executed the same number of times in all its intra-tile wavefronts.

Theorem 2. *Consider a loop nest with $G = (V, E)$ as its DDG. Let $\Phi_{S_1}, \Phi_{S_2}, \ldots$ be its legal tiling hyperplanes. If there exists a $k \in [1, m]$ such that*

$$\phi_S^k(\vec{i_t}) - \phi_S^k(\vec{i_s}) \ge 1, \langle \vec{i_s}, \vec{i_t} \rangle \in \mathcal{P}_{e_{S,S}} \qquad (4)$$

for all self dependences $e_{S,S} \in E$, then the intra-tile wavefronts for every statement S are balanced.

Proof: Follows from Theorem 1 and Definition 1. ■

By combining the constraints given in (4) with those given in (3), $\Phi_S^{\text{intra}} = \left[\begin{smallmatrix}2&1\\1&1\end{smallmatrix}\right]\left[\begin{smallmatrix}t\\i\end{smallmatrix}\right] + \left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$ is found. As illustrated in Figure 6, statement S is executed the same number of times in all intra-tile wavefronts. Therefore, the SPs executing S in a tile are sufficiently utilized with perfect load balance.

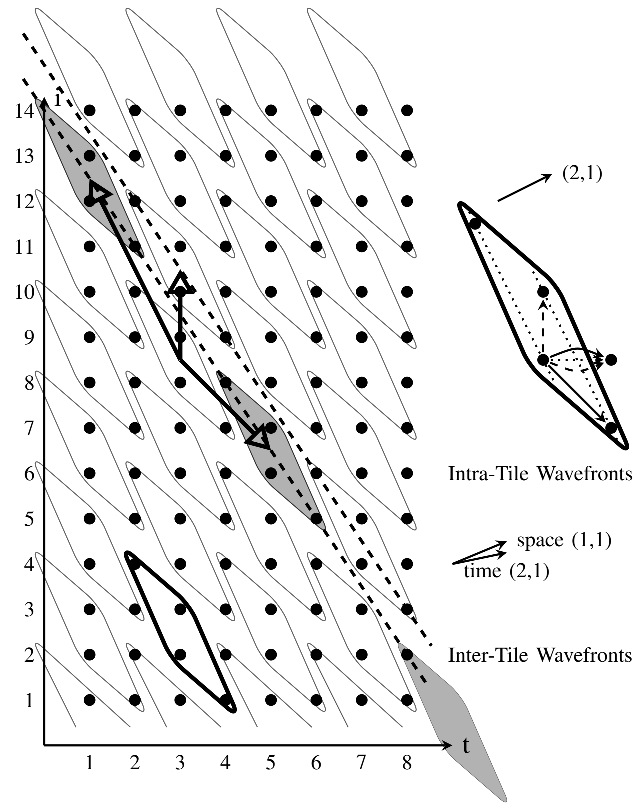


Figure 6: Iteration space tiling of the loop nest in Figure 2 with $\Phi_S^{\text{intra}} = \left[\begin{smallmatrix}2&1\\1&1\end{smallmatrix}\right]\left[\begin{smallmatrix}t\\i\end{smallmatrix}\right] + \left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$. The tile size used is 2×2.

```
Algorithm 1: Generating tiling hyperplanes
  Input: Data dependence graph $G = (V, E)$
  Output: Tiling hyperplanes $\Phi_{S_1}, \Phi_{S_2}, \ldots$
1 Build legality constraints (3);
2 Add constraints for achieving balanced intra-tile
  wavefronts (4) by setting $k = 1$;
3 Find tiling hyperplanes by applying [6], [40], [41];
```

```
A0[I+1]=A[I+1];
for (t=1;t<=T;t++){
    for (i=2;i<=I;i++)
S0:     A0[i]=A[i];
    for (i=1;i<=I;i++)
S1:     A[i]=0.5*(A[i]+A0[i+1]);
}
```

Figure 7: Transformed code by eliminating anti dependence $(0, 1)$ via array copying for the loop nest given in Figure 2.

As shown in Algorithm 1, the tiling hyperplanes can be found by applying the techniques described in [6], [40], [41] except that both (3) and (4) must now be taken into account. This ensures that a loop nest is tiled not only to incur the least inter-tile communication but also to exhibit balanced intra-tile wavefronts for pipelined execution. By setting $k = 1$ in (4), we will generate a set of loops to execute a statement in a tile so that the outermost one iterates over its wavefronts and the remaining loops over its different instances. In GPUs, each thread block is organized as a three dimensional array of threads. In our current implementation, only the innermost three loops in a loop nest are tiled.

C. Improving Inter-Tile Parallelism

By comparing Figure 4 obtained with $\Phi_S = \left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right]\left[\begin{smallmatrix}t\\i\end{smallmatrix}\right]+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$ and Figure 6 obtained with $\Phi_S^{\text{intra}} = \left[\begin{smallmatrix}2&1\\1&1\end{smallmatrix}\right]\left[\begin{smallmatrix}t\\i\end{smallmatrix}\right]+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$, we find that the anti dependence (0,1) affects the shapes of tiles used. In particular, the first hyperplane has changed from (1,0) to (2,1) in order to achieve balanced intra-tile wavefronts when $k = 1$ (Theorem 2). Unfortunately, the price paid is that the tiles become more slanted, reducing the degree of inter-tile parallelism exposed, which can be easily observed by comparing the inter-tile wavefronts in both cases.

In this section, we describe how to eliminate some false (anti and output) dependences in a loop nest by introducing array copy operations. Once they are eliminated in a loop nest, the transformed loop nest can be parallelized with tiles of more “regular” shapes, resulting in improved inter-tile parallelism and reduced bank conflicts. At the same time, balanced intra-tile wavefronts can still be retained.

Definition 2. Consider a loop nest with $G = (V, E)$ as its DDG. A false dependence $e \in E$ is said to be a *parallelism-hindering false dependence* if some constraints generated for

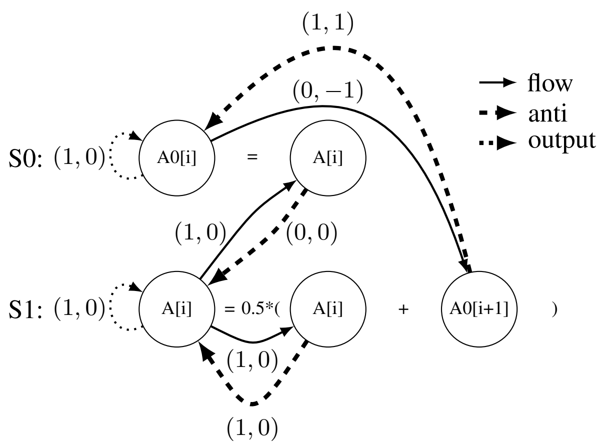


Figure 8: DDG for the loop nest given in Figure 7 with each dependence labelled by its distance vector.

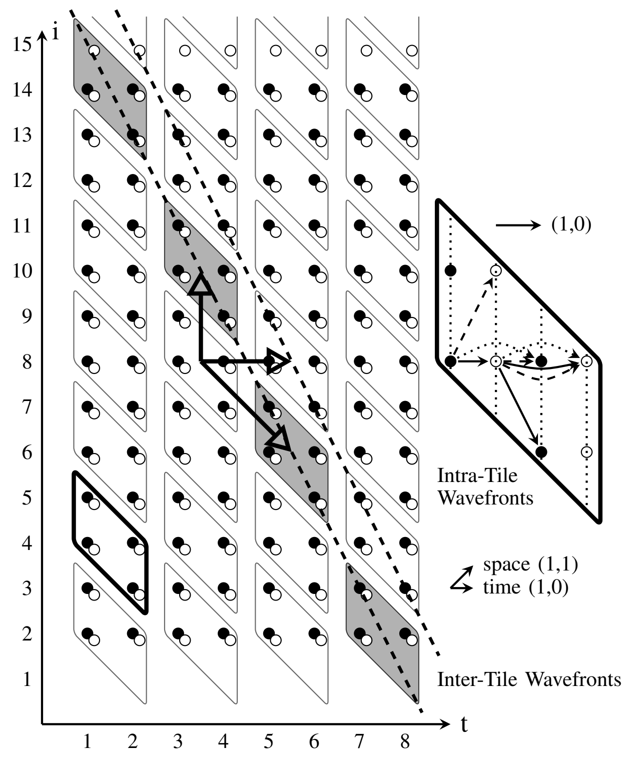


Figure 9: Iteration space tiling of the loop nest in Figure 7 using $\Phi_{S0} = \left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right]\left[\begin{smallmatrix}t\\i\end{smallmatrix}\right]+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$ and $\Phi_{S1} = \left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right]\left[\begin{smallmatrix}t\\i\end{smallmatrix}\right]+\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$. The tile size used is 2×2. The eight dependences depicted inside the larger tile on the right are from Figure 8.

e according to (3) and (4) are *not* redundant with respect to the set of constraints generated for the dependences in $E \setminus \{e\}$ generated according to (3) and (4).

Consider our example with its five dependences listed in

```
A0[I+1]=A[I+1];
/* Inter-tile loops */
for(tt=0;tt<=floor(2*T+I,256);tt++){  //DOSEQ
  for(ii=max(ceil(tt,2),ceil(256*tt-T,256))+blockIdx.x;
      ii<=min(floor(T+I,256),floor(256*tt+I+255,512),tt);ii+=gridDim.x){//DOALL
  /* Intra-tile loops */
    /* Code for coalesced loads from shared memory */
    for(t'=max(1,256*tt-256*ii,256*ii-I);t'<=min(T,256*ii+254,256*tt-256*ii+255);t'++){  //DOSEQ
      for(i'=max(256*ii+1,t'+2)+threadIdx.x;i'<=min(256*ii+256,t'+I);i'+=blockDim.x)  //DOALL
S0:      A0[-t'+i']=A[-t'+i'];
      __syncthreads();
      for(i'=max(256*ii+1,t'+2)+threadIdx.x;i'<=min(256*ii+256,t'+I+1);i'+=blockDim.x)  //DOALL
S1:      A[-t'+i'-1]=0.5*(A[-t'+i'-1]+A0[-t'+i']);
      __syncthreads();
    }
    /* Code for coalesced stores into shared memory */
  }
  /* Code for synchronizing blocks */
}
```

Figure 10: CUDA kernel generated for the code in Figure 7 using the tiling hyperplanes in Figure 9.

Table I and illustrated in Figures 4 and 6. There are three false dependences. The output dependence $(1, 0)$ and the anti dependence $(1, 0)$ are redundant due to the existence of the flow dependence $(1, 0)$. However, the anti dependence $(0, 1)$ is a parallelism-hindering false dependence. This false dependence should be eliminated to improve the load balance during the parallel execution of inter-tile wavefronts.

Following [43], we eliminate the anti dependence $(0, 1)$ by introducing a temporal array A0 to perform some array copy operations. The resulting code is given in Figure 7. To ensure that the anti dependence is not violated, some values of A[i] are copied into A0[i] so that they can be read later when needed. In the transformed code, there are a total of eight dependences as depicted in Figure 8.

There are now two statements, S0 and S1. To find tiling hyperplanes Φ_{S0} and Φ_{S1} with balanced intra-tile wavefronts for both statements, we solve (3) and (4) for all the eight dependences in the transformed loop nest to obtain: $\Phi_{S0} = \left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right]\left[\begin{smallmatrix}t\\i\end{smallmatrix}\right]+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$ and $\Phi_{S1} = \left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right]\left[\begin{smallmatrix}t\\i\end{smallmatrix}\right]+\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$. As a result, the transformed loop nest is tiled as shown in Figure 9. By comparing with Figure 6, we find that the tiles are now more regularly shaped, yielding better inter-tile parallelism. In addition, the intra-tile wavefronts are also balanced.

Furthermore, making tile shapes regular also tends to reduce bank conflicts incurred during the execution of a tile. For loop nests with uniform dependences, the number of such bank conflicts can be easily estimated. This fact can be exploited in tile size selection to tune for better tiling hyperplanes (Section IV). Consider our example again. In Figures 4 and 6, where the anti dependence $(0, 1)$ is still present, the wavefronts in a tile are pipelined along $(2, 1)$. As a result, each wavefront can be identified by a line $i = -2t + c$ for some c. As the values of i are discontinuous with a gap of 2, stride-2 memory accesses are made. When one warp executes a wavefront, two-way bank conflicts will occur. In Figure 9, where the anti dependence $(0, 1)$ has been removed, the memory accesses in each wavefront are now stride-1. As a result, no bank conflicts can occur.

```
Algorithm 2: Eliminating parallelism-hindering false
dependences by introducing array copy operations
  Input: A loop nest $L$ with uniform dependences
  Output: Transformed loop nest $L'$ with the same I/O
          behavior as $L$
1 Identify parallelism-hindering false dependences in $L$;
2 Transform $L$ into $L'$ by eliminating these false
  dependences using array copy operations [43];
```

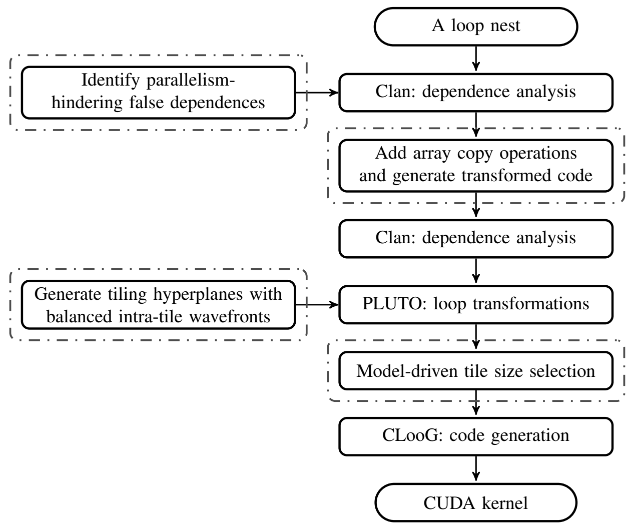
The CUDA code generated using the tiling hyperplanes illustrated in Figure 9 is given in Figure 10. There are two statements S0 and S1. The wavefronts for each statement in each tile are pipelined for parallel execution. Both are synchronized to satisfy their inter-statement dependences.

As shown in Algorithm 2, once parallelism-hindering false dependences in a loop nest are identified, we can apply the techniques described in [43] to eliminate them. This allows the transformed loop nest to be parallelized using tiling hyperplanes with better inter-tile parallelism.

IV. The Compiler Framework

We have implemented our compiler techniques using a combination of the Clan polyhedral representation extractor, PLUTO's polyhedral parallel tiling infrastructure and the CLooG code generator, as shown in Figure 11. Our framework automatically translates sequential C loop nests into CUDA kernels. Our techniques are used in the modules highlighted by the dotted rectangular boxes.

We identify parallelism-hindering false dependences in a loop nest using Clan's dependence analysis. We then



A loop nest
Identify parallelism-hindering false dependences
Clan: dependence analysis
Add array copy operations and generate transformed code
Clan: dependence analysis
Generate tiling hyperplanes with balanced intra-tile wavefronts
PLUTO: loop transformations
Model-driven tile size selection
CLooG: code generation
CUDA kernel

Figure 11: The C-to-CUDA compiler framework.

generate the transformed loop nest without these false dependences by using the techniques described in [43]. When searching for tiling hyperplanes with balanced intra-tile wavefronts and performing subsequent loop transformations, we make use of PLUTO's polyhedral implementation.

We previously developed a cost model regarding tile size selection for GPUs [9]. This model estimates the execution times of a loop nest with different tile sizes and thread organizations. When intra-tile wavefronts are balanced, the predictions made by our model can be more precise.

Finally, we use CLooG with the extension described in [4] to generate CUDA code from the tiling hyperplanes selected.

V. Experimental Results

We have selected eight benchmarks on stencil computations, as listed in Table II. These include 1-D, 2-D and 3-D Jacobi solvers from the Rodinia benchmark suite and the Circular Queue toolkit, 2-D FDTD as well as 1-D and 2-D SOR solvers from Polybench, and the 2-D Heat solver from the Chombo toolkit. For the two 3-D Jacobi solvers, only their innermost three loops are tiled and parallelized. Our evaluation with small benchmarks is preliminary. In future work, more extensive benchmarking will be conducted.

We have carried out our experiments on two NVIDIA GPUs, C1060 and C2050. For each benchmark, the problem size used is given in Table II. For all the benchmarks, we have succeeded in finding tiling hyperplanes with balanced intra-tile wavefronts. For all except the two SOR solvers, some parallelism-hindering false dependences have been eliminated to expose more inter-tile parallelism.

We evaluate our framework by comparing with PLUTO [4], an existing C-to-CUDA translator, and some hand-optimized CUDA code. We compare our framework and PLUTO across all the eight benchmarks to highlight the performance advantages of the two new optimizations introduced for GPUs in this paper. We compare the CUDA code generated for the four Jacobi solvers with their highly hand-tuned CUDA kernels to demonstrate further the strengths and limitations of our compiler optimizaitons.

1) Compared with PLUTO: Both our framework and PLUTO generate tiling hyperplanes for loop nests automatically. In both cases, the best tile sizes for tiling hyperplanes are determined empirically by using a cost model from [9].

Figure 12 shows the speedups achieved by our framework over PLUTO. There are two configurations tested in our framework, depending on how the two optimizations listed in Table II are used. "Intra" means that the optimization for achieving balanced intra-wavefronts is turned on. "Intra+Inter" means that in addition to the "intra" optimization, the "inter" optimization for eliminating parallelism-hindering false dependences to achieve better inter-tile parallelism is also turned on. There are no such false dependences to eliminate in the two SOR solvers. Therefore, the speedups in the two configurations in either benchmark are identical. The "Inter" optimization, if used alone, is not very beneficial. Figure 13 shows the speedups on C2050.

Our experimental results demonstrate the performance advantages of the "intra" and "inter" optimizations. When the "intra" optimization alone is used, the speedups range from 2.05X – 5.41X with an average of 3.32X on C1060 and from 2.07X – 5.43X with an average of 3.27X on C2050. As the intra-tile wavefronts are balanced, the degree of intra-SM parallelism achieved in our framework is higher.

When the "inter" optimization is also turned on, there are performance improvements across all the six benchmarks (with some parallelism-hindering false dependences to remove) on both C1060 and C2050. The speedups range from 3.10X – 5.52X with an average of 3.98X on C1060 and from 3.20X – 5.48X with an average of 4.07X on C2050. By eliminating some false dependences, tiles with more regular shapes can be used. This optimization has two benefits. First, better inter-tile parallelism is achieved. Second, the number of bank conflicts incurred during the execution of a tile is reduced. On the other hand, this optimization also has its associated costs. False dependences are eliminated by introducing new temporary arrays and new copy operations on these arrays. Such code rewriting can increase the number of instructions executed and inter-statement synchronization operations used. Overall, the benefits outweigh the costs.

2) Compared with Hand-tuned Code: We compare the CUDA code generated by our framework for the four Jacobi benchmarks with the CUDA code manually obtained using the Circular Queue approach [8]. Circular Queue improves data locality by streaming tiled data. It strives to exploit the maximum amount of parallelism among the computations executed during every iteration of the outermost (time) loop but ignores the data reuse across its different iterations. Unlike Circular Queue, our approach trades off parallelism

Benchmark	Max Loop Depth	Innermost DOACROSS Loop?	Perfectly Nested?	Our Approach: False Dependence Elimination	Our Approach: Balanced Intra-Tile Wavefronts	Input Problem Size
1-D Jacobi-3 (3 points)	2			✓	✓	65536*65536
2-D Jacobi-5 (5 points)	3			✓	✓	1000*4096*4096
3-D Jacobi-7 (7 points)	4			✓	✓	256*256*256*256
3-D Jacobi-27 (27 points)	4			✓	✓	256*256*256*256
1-D SOR-3 (3 points)	2	✓	✓		✓	65536*65536
2-D SOR-5 (5 points)	3	✓	✓		✓	1000*4096*4096
2-D FDTD	3			✓	✓	1000*4096*4096
2-D Heat (7 points)	3			✓	✓	1000*4096*4096

Table II: Benchmarks and their characteristics.

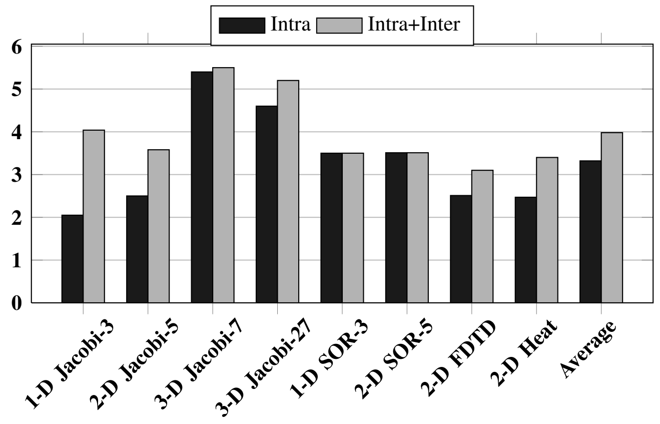


Figure 12: Speedups of our framework (with the two optimization configurations) over PLUTO on C1060.

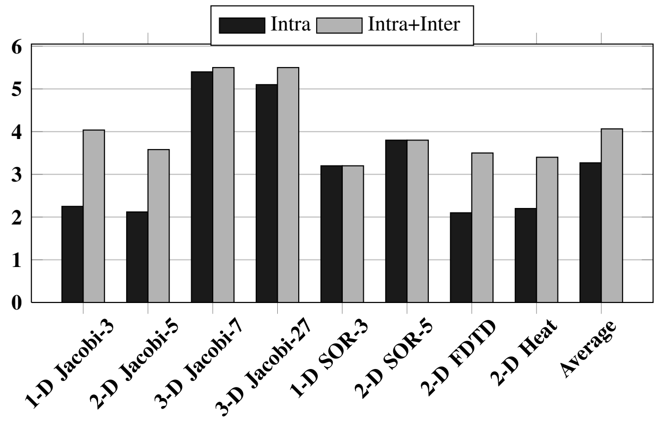


Figure 13: Speedups of our framework (with the two optimization configurations) over PLUTO on C2050.

for data reuse due to pipelined execution of wavefronts.

As shown in Figure 14, our approach is competitive. For 1-D Jacobi (3 points), 2-D Jacobi (5 points), 3-D Jacobi (7 points) and 3-D Jacobi (27 points), our speedups are

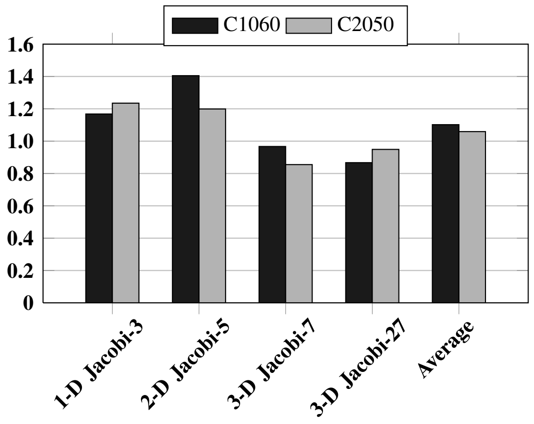


Figure 14: Speedups over Circular Queue for Jacobi.

1.17X, 1.41X, 0.97X and 0.87X with an average of 1.10X on C1060 and 1.24X, 1.20X, 0.86X and 0.95X with an average of 1.06X on C2050. For the 1-D and 2-D solvers, we achieve better performance due to better data reuse exploited. For the two 3-D solvers, only the innermost three loops are parallelized. Our CUDA kernels are similar to the ones generated by Circular Queue. Circular Queue delivers better performance due to some hand optimizations. For example, simpler loop bounds are used, resulting in fewer instructions to be executed. The amount of data transferred between device memory and shared memory is meticulously calculated. Finally, memory accesses are better coalesced.

VI. Related Work

A. Performance Analysis and Compilation for GPUs

Due to complexities involved in optimizing GPU applications, several studies have been conducted to establish optimization principles and strategies that allow efficient mapping of sequential programs to GPUs [3], [15], [34]. There are two major issues affecting the productivity bottlenecks in GPU programming: utilization of memory hierarchy and management of parallelism. To address these two issues, G-

ADAPT [23] was introduced as an input-adaptive optimization framework for GPUs to predict optimal configurations. Yang et al. [44] designed an optimizing compiler to alleviate these bottlenecks. In addition, OpenMPC [18], [19] and OMPSs [12] provide an abstraction for the complex CUDA programming model and offer high-level controls over the involved parameters and optimizations.

Manual development of efficient CUDA code for different loop nests can be time-consuming and error-prone. In order to reduce programming effort, PLUTO [4] was released as a source-to-source compiler for translating sequential C programs into CUDA programs. CETUS [19] and PGI [39] support high-level programming paradigms for GPUs, which are similar to the widely-used OpenMP programming model. Par4All [28] combines C-to-OpenMP and OpenMP-to-CUDA to generate CUDA programs. While performing some code optimizations, these tools do not specifically optimize loop nests with loop-carried dependences, as we do.

B. Stencil Computations on GPUs

There are a number of prior efforts on implementing stencil kernels on GPUs [8], [16], [22], [25], [32], [37]. In the case of Jacobi-like stencil computations with DOALL inner loops, Circular Queue [8] streams tiled data to hide I/O latency and exploit data locality in the GPU memory hierarchy. However, it does not exploit the data reuse among multiple sweeps across a computational domain [42]. Our approach does this and achieves better performance than Circular Queue for 1-D and 2-D Jacobi solvers.

In [22], [37], efficient "asynchronous" solutions for Jacobi are introduced by not respecting some loop-carried dependences. The basic idea is to algorithmically restructure a stencil kernel based on a non-dependence-preserving parallelization scheme to avoid pipelining for higher parallelism. In contrast, our approach exploits better fine-grained parallelism without violating data dependences and is applicable to all loop nests with uniform dependences.

C. Multi-level Tiling in the Polyhedral Model

Multi-level tiling has been developed to improve data locality across different levels of a memory hierarchy [14], [33], [35]. This work aims to achieve two levels of fine-grained parallelism for GPUs: intra-SM and inter-SM. We improve intra-SM parallelism by generating balanced intra-tile wavefronts. We improve inter-SM parallelism by eliminating parallelism-hindering false dependences.

When tiling a loop nest, there are infinitely many choices. To enable appropriate tiling hyperplanes to be selected, some work focuses on pruning the search space and optimizing hyperplane generation strategies [29], [30], [45], [42]. Lim and Lam's algorithm obtains affine partitions that minimize the order of communication while maximizing the degree of parallelism [21]. Griebl's approach enables tiling of the time dimension with a forward communication-only placement [13]. Bondhugula [6] developed a *communication-minimized parallelization* framework in which a cost function is used to quantify communication volumes and reuse distances.

D. Dependence Elimination

A common problem faced in restructuring programs is how to suppress anti and output dependences in order to improve locality [43] or parallelism [20], [24]. This is usually done by array expansion or introducing new temporary arrays and associated copy operations [7], [11], [35], [43].

Unlike these previous efforts, which are restricted to CPUs, this work eliminates anti and output dependences to improve fine-grained (intra- and inter-SM) parallelism on GPUs. Such false dependences are eliminated in order to improve inter-tile parallelism and reduce bank conflicts.

VII. Conclusion

We have presented a compiler framework that enables automatic parallelization of loop nests into CUDA code with enhanced fine-grained parallelism for GPUs. Our framework comprises two key optimizations, one to find tiling hyperplanes with balanced intra-tile wavefronts and one to eliminate some false dependences with improved inter-tile parallelism. Our preliminary experimental validation demonstrates the effectiveness of our framework.

Acknowledgment

We thank the reviewers for their comments and suggestions. This work is supported by the Australian Research Council Grants, DP0881330 and DP110104628.

References

[1] M. Amini, F. Coelho, F. Irigoin, and R. Keryell. Static compilation analysis for host-accelerator communication optimization. In *LCPC*, 2011.

[2] S. Baghdadi, A. Größ linger, and A. Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *CPC*, 2010.

[3] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp, and W. Hwu. An adaptive performance modeling tool for GPU architectures. In *PPoPP*, pages 105–114, 2010.

[4] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC*, pages 244–263, 2010.

[5] U. Bondhugula. *Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model*. PhD thesis, Ohio State University, 2010.

[6] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *CC*, pages 132–146, 2008.

[7] P. Calland, A. Darte, Y. Robert, and F. Vivien. On the removal of anti and output dependences. *International Journal of Parallel Programming*, 26(2):285–312, 1998.

[8] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, pages 1–12, 2008.

[9] P. Di and J. Xue. Model-Driven Tile Size Selection for DOACROSS Loops on GPUs. In *Euro-Par*, pages 401–412, 2011.

[10] H. Dursun, K. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R.K. Kalia, A. Nakano, and P. Vashishta. In-core optimization of high-order stencil computations. In *PDPTA*, pages 533–538, 2009.

[11] P. Feautrier. Array Expansion. In *ICS*, pages 429–441, 1988.

[12] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. González, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL. In *LCPC*, pages 215–229, 2010.

[13] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. PhD thesis, University of Passau, 2004.

[14] A. Hartono, M.M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *ICS*, pages 147–157, 2009.

[15] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, pages 152–163, 2009.

[16] X. Huo, V. Ravi, W. Ma, and G. Agrawal. An execution strategy and optimized runtime support for parallelizing irregular reductions on modern GPUs. In *ICS*, pages 2–11, 2011.

[17] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. Strout. Multi-level Tiling: M for the Price of One. In *SC*, pages 1–12, 2007.

[18] S. Lee and R. Eigenmann. OpenMPC : Extended OpenMP Programming and Tuning for GPUs. In *SC*, pages 1–11, 2010.

[19] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP*, pages 101–110, 2009.

[20] Z. Li. Array privatization for parallel execution of loops. In *ICS*, pages 313–322, 1992.

[21] A. Lim and M. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998.

[22] L. Liu and Z. Li. Improving parallelism and locality with asynchronous algorithms. In *PPoPP*, pages 213–222, 2010.

[23] Y. Liu, E. Zhang, and X. Shen. A Cross-Input Adaptive Framework for GPU Programs Optimization. In *IPDPS*, pages 16–19, 2009.

[24] D. Maydan, S. Amarsinghe, and M. Lam. Data dependence and data-flow analysis of arrays. In *LCPC*, pages 434–448, 1993.

[25] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *ICS*, pages 256–265, 2009.

[26] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *SC*, pages 1–13, 2010.

[27] NVIDIA. NVIDIA CUDA Programming Guide 3.0, 2010.

[28] Par4All. http://www.par4all.org, 2012.

[29] E. Park, L. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *CGO*, pages 119–129, 2011.

[30] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: convexity, pruning and optimization. In *POPL*, volume 46, pages 549–562, 2011.

[31] A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer, 1994.

[32] F. Rahman, Q. Yi, and A. Qasem. Understanding stencil code performance on multicore architectures. In *CF*, pages 30:1–30:10, 2011.

[33] L. Renganarayanan. *Scalable and Efficient Tools for Multi-level Tiling*. PhD thesis, Colorado State University, 2008.

[34] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, pages 73–82, 2008.

[35] M. Strout. *Performance Transformations for Irregular Applications*. PhD thesis, University of California, San Diego, 2003.

[36] J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *CoRR*, abs/1004.1741, 2010.

[37] S. Venkatasubramanian and R. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *ICS*, pages 244–255, 2009.

[38] M. Wittmann, G. Hager, J. Treibig, and G. Wellein. Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters. *CoRR*, abs/1006.3148, 2010.

[39] M. Wolfe. Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on GPU*, pages 43–50, 2010.

[40] J. Xue. Communication-Minimal Tiling of Uniform Dependence Loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.

[41] J Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.

[42] J. Xue and C. Huang. Reuse-driven tiling for data locality. In *LCPC*, pages 16–33, 1997.

[43] J. Xue and Q. Huang. Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences. In *ICPP*, pages 107–115, 2005.

[44] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, pages 86–97, 2010.

[45] Q. Yi. POET: A Scripting Language For Applying Parameterized Source-to-source Program. *Softw. Pract. Exper.*, 42(6):675–706, 2012.