Model-Driven Tile Size Selection for DOACROSS Loops on GPUs*

Peng Di and Jingling Xue

Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Abstract. DOALL loops are tiled to exploit DOALL parallelism and data locality on GPUs. In contrast, due to loop-carried dependences, DOACROSS loops must be skewed first in order to make tiling legal and exploit wavefront parallelism across the tiles and within a tile. Thus, tile size selection, which is performance-critical, becomes more complex for DOACROSS loops than DOALL loops on GPUs. This paper presents a model-driven approach to automating this process. Validation using 1D, 2D and 3D SOR solvers shows that our framework can find the tile sizes for these representative DOACROSS loops to achieve performances close to the best observed for a range of problem sizes tested.

1 Introduction

GPGPUs have become one of the most powerful and popular platforms to exploit fine-grain parallelism in high performance computing. Recent research on developing programming and compiler techniques for GPUs focuses on (among others) general programming principles [5,9,14], cost modeling and analysis [1,15,3], automatic code generation [2,11], and performance tuning and optimization [4,6,12,19]. However, these research efforts are almost exclusively limited to DOALL loops. In practice, DOACROSS loops play an important role in many scientific and engineering applications, including PDE solvers [13], efficient preconditioners [7] and robust smoothers [8]. Presently, Pluto [2] seems to be the only framework that can map sequential DOACROSS loops to CUDA code automatically for NVIDIA GPUs. This is done by applying loop skewing and tiling with user-supplied tile sizes (for a user-declared grid of thread blocks of threads).

DOALL loops are tiled to exploit DOALL parallelism and data locality on GPUs. Unlike DOALL loops, DOACROSS loops must be skewed first to ensure that the subsequent tiling transformation preserves the loop-carried dependences. Furthermore, performing skewing and tiling allows wavefront parallelism to be exploited both across the tiles and within a tile. Tile size selection, which is performance-critical on GPUs, are more complex for DOACROSS than DOALL loops due to parallelism-inhibiting loop-carried dependences and more complex interactions among the GPU architectural constraints. Thus, it is not practical

* This research is supported by an Australian Research Council Grant DP110104628.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 401–412, 2011.

to rely on the user to pick the right tile sizes to optimize code through improving processor utilization and reducing synchronization overhead. Existing tile size techniques proposed for caches in CPU architectures do not apply [10,20].

This paper makes the following contributions:

- We present (for the first time) a model for estimating the execution times of tiled DOACROSS loops running on GPUs (Section 3);
- We introduce a model-driven framework to automate tile size selection for tiled DOACROSS loops running on GPUs (Section 4);
- We evaluate the accuracy of our model using representative 1D, 2D and 3D SOR solvers and show that the tile sizes selected lead to the performances close to the best observed for a range of problem sizes tested (Section 5).

2 Parallelization of DOACROSS Loops on GPUs

We describe a scheme for mapping sequential DOACROSS loops to CUDA code on GPUs. Our illustrating example is a 1D SOR-like solver. This scheme is the same as that supported by Pluto [2] except tiles are mapped to thread blocks in a different way in order to achieve better load balance.

```
for(i1=1;i1<=I1;i1++)
  for(i2=1;i2<=I2;i2++)
    A[i2]=(A[i2-1]+A[i2]+A[i2+1])/3;
```

Fig. 1. Sequential loop nest for the 1D SOR solver

Loop Transformations. In Pluto, parallelizing an n-dimensional DOACROSS loop nest L consists of mapping it into a $2n$-dimensional loop nest as follows:

$$\rho : \mathbb{Z}^n \mapsto \mathbb{Z}^{2n},\ \rho(\boldsymbol{i}) = \begin{pmatrix} \boldsymbol{t} \\ \boldsymbol{e} \end{pmatrix} = \left(t_1, \ldots, t_n, e_1, \ldots, e_n\right)^T = \begin{pmatrix} \mathcal{W}\lfloor \mathcal{T}\mathcal{S}(\boldsymbol{i}) \rfloor \\ \mathcal{W}\mathcal{S}(\boldsymbol{i}) \end{pmatrix} \quad (1)$$

$$\mathcal{T} = \begin{bmatrix} m_1 & 0 & \ldots & 0 \\ 0 & m_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & m_n \end{bmatrix}_{n \times n}^{-1}, \quad \mathcal{W} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 & 1 \\ 0 & 1 & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 0 & 1 \end{bmatrix}_{n \times n} \quad (2)$$

The mapping process for the 1D SOR solver in Figure 1 is illustrated in Figure 2. The mapping ρ [18] is realized by composing a loop skewing $\mathcal{S}$, a loop tiling $\mathcal{T}$ and another loop skewing $\mathcal{W}$. First, the iteration space of L is skewed by a unimodular transformation $\mathcal{S} \in \mathbb{Z}^{n \times n}$. Second, the skewed iteration space is tiled into n-dimensional rectangles of size $m_1 \times \cdots \times m_n$ by $\mathcal{T}$. $\mathcal{S}$ is chosen to guarantee the legality of tiling so that all loop-carried dependences in L are preserved [18]. At this point, a $2n$-dimensional loop nest is created such that the first n loops (called *tile loops*) enumerate the tiles and the inner n loops (called *element loops*) enumerate the iterations within a tile. Finally, another skewing

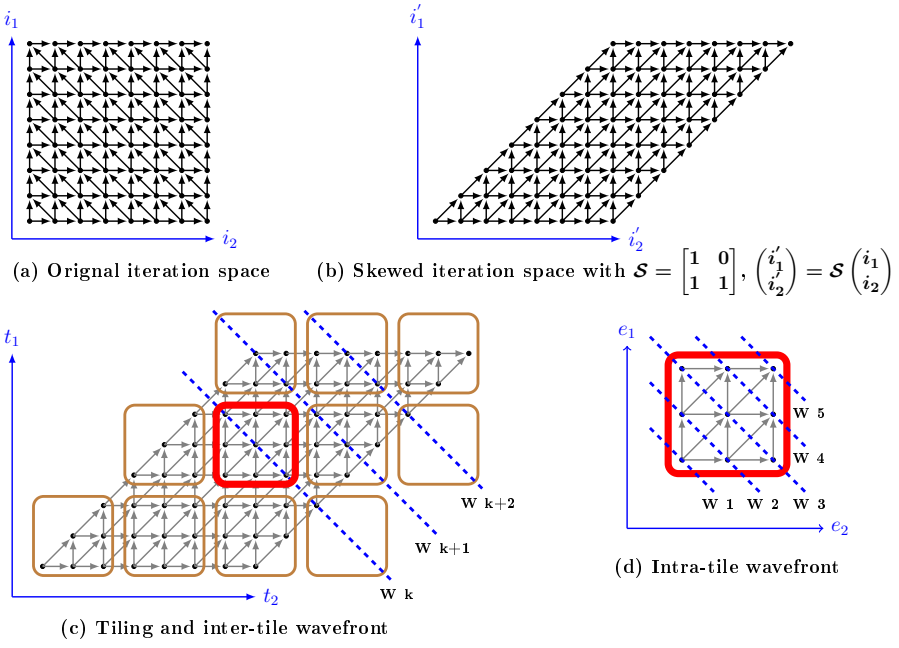

Fig. 2. Exploiting wavefront parallelism for 1D SOR on GPUs

transformation $\mathcal{W}$ is applied to the iteration spaces of both sets of loop nests to expose wavefront parallelism across the tiles and within a tile. In either loop nest, the first loop is sequential and the remaining $n-1$ loops are DOALL. We will speak of inter-tile wavefronts and intra-tile wavefronts (as shown in Figure 2).

Mapping to GPUs. A NVIDIA GPU consists of a number of *streaming multiprocessors (SMs)*, each of which contains a number of processor cores called *streaming processors (SPs)*. All SPs in one SM share a local memory and a set of registers. GPU programming is enabled through CUDA. A kernel is executed by a grid of threads organized as *thread blocks* (known as a *thread organization*). A thread block is scheduled to execute on any one of the SMs (as a whole). The threads in a block are partitioned into 32-thread *warps*, which are units of execution (on the SPs of the SM to which the block is mapped). The threads in one block can synchronize through `syncthreads` and communicate through shared memory. The inter-block synchronization is not directly supported.

Figure 3 gives the CUDA code for the 1D SOR solver parallelized as shown in Figure 2. Tiles are mapped to thread blocks and individual loop iterations in a tile are mapped to the threads in a block. All inter-tile wavefronts are executed sequentially to satisfy inter-tile (or inter-block) dependences. Hence, the `syncblocks` macro as introduced in Pluto at ④. In Pluto, the tiles in an $(n-1)$-dimensional inter-tile wavefront are distributed over a 2D grid of thread blocks of size `gridDim.x` × `gridDim.y` cyclically along two of the $n-1$ dimensions

```
// inter-tile loop nest
for(t1=Lt1;t1<=Ut1;t1++){
  for(t2=Lt2(t1)+blockIdx.x;t2<=Ut2(t2);t2+=gridDim.x){
    // intra-tile loop nest
    Code for shared memory coalesced loads
    __syncthreads();  // ① barrier for the loads
    for(e1=Le1(t1,t2);e1<=Ue1(t1,t2);e1++){
      for(e2=Le2(t1,t2,e1)+threadIdx.x;e2<=Ue2(t1,t2,e1);e2+=blockDim.x){
        i2=h(t1,t2,e1,e2);
        A[i2]=(A[i2-1]+A[i2]+A[i2+1])/3;
      }
      __syncthreads();  // ② barrier for each intra-tile wavefront
    }
    Code for shared memory coalesced stores
    __syncthreads();  // ③ barrier for the stores
  }
  __syncblocks();  // ④ barrier for each inter-tile wavefront
}
```

DOALL

Fig. 3. CUDA code for 1D SOR on GPUs

of the wavefront. This can cause load imbalance for large tiles since a wavefront has irregular boundaries. In this paper, the tile coordinates in such a wavefront are "linearized" much like how the subscripts of a multi-dimensional array are. Then the tiles are mapped to a 1D grid of thread blocks of size `gridDim.x` cyclically to achieve better load balance (with `gridDim.y`=1 always). As a result, all thread blocks in an inter-tile wavefront can be potentially executed in parallel but the tiles within a block are always executed sequentially.

The loop iterations in a tile are distributed as in Pluto to a 3D thread block of size `blockDim.x` × `blockDim.y` × `blockDim.z` cyclically. To improve memory coalescing, all data read by a tile are first loaded from device memory to shared memory at ① and all those written in a tile are stored back to device memory at ③. Like inter-tile wavefronts, all intra-tile wavefronts are executed sequentially. Hence, the `syncthreads` instruction at ②. The iterations in an intra-tile wavefront that are assigned to different threads can execute in parallel.

3 Execution Time Modeling

We parameterise an execution time model for a tiled DOACROSS loop nest in order to automate tile size selection. Initially, we assume that all tiles are full. We consider first intra-tile execution (Section 3.1) and then inter-tile execution (Section 3.2). In Section 3.3, we estimate the parameters used. In Section 3.4, we discuss briefly how to mitigate the effects of border tiles on performance.

3.1 Intra-tile Execution

This section focuses on estimating the execution time, $T_{\mathcal{TILE}}$, for a single (full) tile, denoted $\mathcal{TILE}$. As shown in (1) and Figures 2 and 3, the loop iterations in a tile are indexed by $(e_1, \ldots, e_n)$, where e_1 enumerates all intra-wavefronts within the tile. As illustrated in Figure 2, $T_{\mathcal{TILE}}$, which can be broken down into the time on loading the input data at ①, the time on executing the tile, and the time on storing the results back ③, is approximated by:

$$T_{\mathcal{TILE}} = \sum_{e_1=L_{e1}}^{U_{e1}} T_{e1} + T_{mem} + 2\sigma_{thd} \tag{3}$$

The first term $\sum_{e_1=L_{e1}}^{U_{e1}} T_{e1}$ is the computation cost of $\mathcal{TILE}$ estimated as a sum of the execution times T_{e1} of all its intra-tile wavefronts with $e1$ ranging over these wavefronts starting from the smallest given by the lower bound L_{e1} of loop $e1$ to the largest given by the upper bound U_{e1} of loop $e1$ along dimension $e1$. The second term T_{mem} denotes the memory latency consumed by the memory accesses at the code before ① and the code before ③. The last term $2\sigma_{thd}$ denotes the overhead of the two `syncthreads` at ① and ③, where σ_{thd} is dependent on the number of threads used, i.e., `blockDim.x` × `blockDim.y` × `blockDim.z`.

The execution time T_{e1} of the intra-tile wavefront indexed by $e1$ is given by:

$$T_{e1} = \alpha \times G_{e1} + \beta \times H_{e1} + \sigma_{thd} \tag{4}$$

GPUs execute instructions with warps as units of execution and hide memory latency through interleaving of thread blocks. In the scheme shown in Figure 3, the warps are never idle when executing a wavefront as all memory accesses happen before and after the execution of $\mathcal{TILE}$. Thus, the first term represents the workload for computing the wavefront indexed by e_1, which is estimated to be proportional to G_{e1}, the number of 32-thread warps executed at the wavefront. In addition, the first term implicitly considers the effects of bank conflicts on the execution time of $\mathcal{TILE}$. However, the same G_{e1} may result when the number of loop iterations, H_{e1}, in the wavefront indexed by $e1$ varies (due to division by 32). To accommodate its impact on performance, T_{e1} is fine-tuned by including the second term $\beta \times H_{e1}$, which attempts to differentiate the effects of varying H_{e1} values on performance. Note that G_{e1} and H_{e1} may vary from wavefront to wavefront as shown in Figure 2(d). Given an intra-tile wavefront, both can be precisely calculated. The last term σ_{thd} is the overhead of `syncthreads` at ②.

By substituting T^i_{e1} in (4) into (3), we obtain:

$$T_{\mathcal{TILE}} = \sum_{e1=L_{e1}}^{U_{e1}} (\alpha \times G_{e1} + \beta \times H_{e1} + \sigma_{thd}) + T_{mem} + 2\sigma_{thd} \tag{5}$$

which is illustrated graphically in Figure 4. As highlighted, G_{e1} varies across the wavefronts with less work being done at the beginning and end of the computation process for $\mathcal{TILE}$ when its wavefronts are executed.

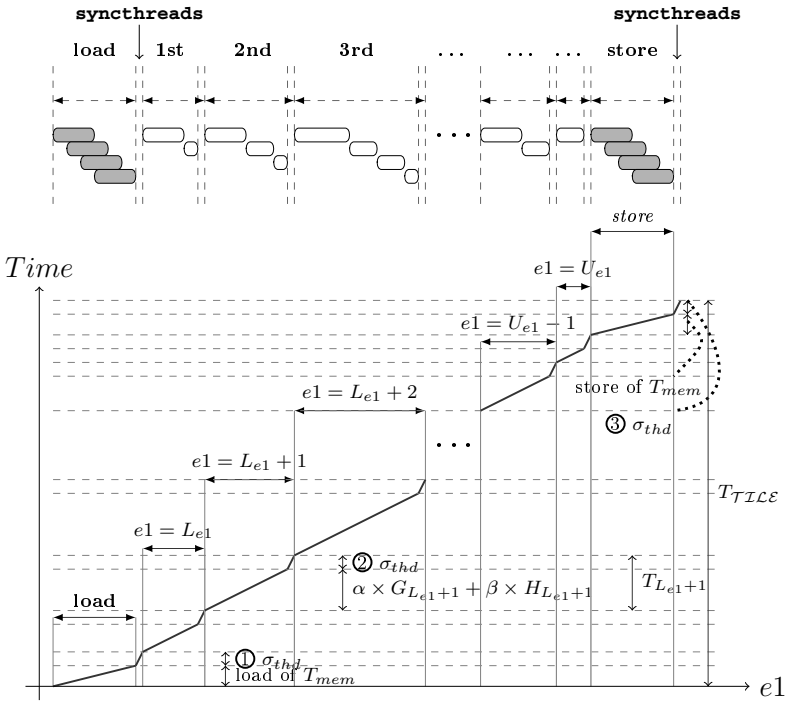

Fig. 4. Execution of the N_{e1} wavefronts of $\mathcal{TILE}$ along dimension e_1 according to (5)

The memory access latency T_{mem} can be estimated by:

$$T_{mem} = \gamma \times N_{mem} \tag{6}$$

where N_{mem} denotes the number of loads and stores made in $\mathcal{TILE}$. Note that N_{mem} is not related to memory coalescing since that would make its estimation dependent on the actual data layout at run time. This simple approximation seems to be adequate as T_{mem} is not a dominant term in (3) for the following reasons combined. First, DOACROSS loops are usually not bandwidth-bound. Second, optimal tile sizes found are large in order to exploit two levels of wavefront parallelism. Finally, the memory accesses performed by warps can overlap. We will return to this issue briefly at the end of Section 5.

By substituting T_{mem} given in (6) into (5), simplifying and letting

$$L_{\mathcal{TILE}} = m_1 \times \cdots \times m_n = \sum_{e1=L_{e1}}^{U_{e1}} H_{e1} \tag{7}$$

we obtain the following estimated execution time of $\mathcal{TILE}$:

$$T_{\mathcal{TILE}} = \alpha \times \sum_{e1=L_{e1}}^{U_{e1}} G_{e1} + (U_{e1} - L_{e1} + 3) \times \sigma_{thd} + \beta \times L_{\mathcal{TILE}} + \gamma \times N_{mem} \tag{8}$$

with $U_{e1} - L_{e1} + 1$ `syncthreads` instructions executed at ② inside the wavefront, one at ① and one at ③, as shown in Figure 3.

3.2 Inter-tile Execution

A DOACROSS loop nest is parallelized into a CUDA kernel. The execution time, T_{total}, for the kernel, i.e., for its inter-tile wavefronts is estimated by:

$$T_{total} = \sum_{t1=L_{t1}}^{U_{t1}} T_{t1} + \sigma_{ker} \tag{9}$$

The first term is the computation cost of all tiles in the kernel estimated as a sum of the execution times T_{t1} of all its inter-tile wavefronts with $t1$ starting from the lower bound L_{t1} of loop $t1$ to the upper bound U_{t1} of loop $t1$ along dimension $t1$. The second term σ_{ker} is the kernel startup cost.

Thus, $U_{t1} - L_{t1} + 1$ is the number of tiles contained in the inter-tile wavefront indexed by $t1$. If all P *SMs* execute simultaneously up to B thread blocks each, then the number of tiles, denoted I_{t1}, contained in a thread block is:

$$I_{t1} = \lceil \frac{U_{t1} - L_{t1} + 1}{B \times P} \rceil = \lceil \frac{U_{t1} - L_{t1} + 1}{\texttt{gridDim.x}} \rceil \tag{10}$$

where B is decided by the GPU architectural constraints and kernel code according to the CUDA programming guide as demonstrated in Table 1.

The execution time T_{t1} is determined by the slowest among the P SMs with the other SMs idle waiting at the `syncblocks` macro at ④. As a result, we have:

$$T_{t1} = \sum_{i=1}^{I_{t1}} T_{t1}^{i} + \sigma_{blk} \tag{11}$$

$$T_{t1}^{i} = \begin{cases} B \times T_{\mathcal{TILE}} & 1 \le i \le I_{t1} - 1 \\ \lceil \frac{(U_{t1} - L_{t1} + 1)\%(B \times P)}{P} \rceil \times T_{\mathcal{TILE}} & i = I_{t1} \end{cases} \tag{12}$$

where T_{t1}^{i} is the execution time of the i-th tiles in all B thread blocks by the slowest SM and σ_{blk} is the overhead of `syncblocks` at ④ (to be measured below).

3.3 Parameter Estimation

We determine the six parameters used in T_{total} given in (9) for a tiled loop nest L: σ_{ker}, σ_{thd}, σ_{blk}, α, β and γ. We do so for a given thread organization (determined by `gridDim` and `blockDim`) so that its tile size selection can be automated (Section 4). For NVIDIA GPUs, there are at most $16B_{max}$ different thread organizations because (1) there are B_{max} different 1D grid layouts with $\texttt{gridDim.x} = B \times P$, where $B \le B_{max} \le B_{SM} = 8$ as shown in Table 1, and (2) the number of threads per block, i.e., $\texttt{blockDim.x} \times \texttt{blockDim.y} \times \texttt{blockDim.z}$ is one of the 16 possibilities contained in $\{32, 64, \ldots, 512\}$.

Table 1. Determining B for an NVIDIA Tesla C1060 GPU. An item in Column 1 that depends on hardware, kernel code or both is indicated with an H, S or B appropriately.

Description	Name
Warp Size (H)	$W = 32$
Max Number of Active Warps per SM (H)	$W_{SM} = 32$
Max Number of Active Threads per SM (H)	$T_{SM} = 1024$
Max Number of Active Blocks per SM (H)	$B_{SM} = 8$
Shared Memory per SM (H)	$S_{SM} = 16KB$
Number of 32-bit Registers per SM (H)	$R_{SM} = 16K$
Threads per Thread Block (S)	K
Register Usage per Thread Block (S)	R_{TB}
Shared Memory Usage per Thread Block (S)	S_{TB}
Warps per Thread Block (B)	$W_{TB} = \lceil \frac{K}{W} \rceil$
Thread Blocks Limited by Warps (B)	$B_W = \min(B_{SM}, \lfloor \frac{W_{SM}}{W_{TB}} \rfloor)$
Thread Blocks Limited by Registers (B)	$B_R = \lfloor \frac{R_{SM}}{R_{TB}} \rfloor$
Thread Blocks Limited by Shared Memory (B)	$B_S = \lfloor \frac{S_{SM}}{S_{TB}} \rfloor$
Thread Blocks (B)	$B = \min(B_W, B_R, B_S)$

Architectural Parameters: σ_{ker}, σ_{thd} and σ_{blk}. These overheads are small (relative to the execution time of a loop nest L) and are measured for a GPU architecture as follows. First of all, σ_{ker} is the startup overhead of the kernel for L, which can be obtained through running an empty version of the kernel (with the computations in L removed) for a given thread organization. In fact, as $\sigma_{ker} \ll T_{total}$, treating it as a small constant for all thread organizations does not affect in practical terms how the relative performances of L are ranked for all combinations of thread organizations and tile sizes used. As for `syncthreads` executed at ①, ② and ③, it is lightweight on NVIDIA GPUs. Its overhead σ_{thd} depends on the number of threads per block, i.e., `blockDim.x` × `blockDim.y` × `blockDim.z` and is measured as in [16]. There are only 16 cases to consider as `blockDim.x` × `blockDim.y` × `blockDim.z` is a multiple of 32 ranging from 32 to 512. Finally, the `syncblocks` macro is invoked at the end of each inter-tile wavefront at ④. Its overhead σ_{blk}, which is higher than σ_{thd}, depends mainly on the number of thread blocks contained in an inter-tile wavefront, i.e., `gridDim.x` $= B \times P$. The effects of different `blockDim.x` × `blockDim.y` × `blockDim.z` values on σ_{blk} are negligible. As $B \leq B_{max} \leq B_{SM} = 8$, `syncblocks` is measured as in [17] for a few, i.e., up to B_{max} different `gridDim.x` values.

Program-Dependent Parameters: α, β and γ. Once the values of σ_{ker}, σ_{thd} and σ_{blk} are determined, the given loop nest L is simplified to possess one inter-tile wavefront with exactly $B \times P$ thread blocks consisting of only full tiles. This ensures that all P SMs have exactly the same workload so that these three program-dependent parameters can be accurately measured.

The three parameters are found for each of up to $16B_{max}$ different thread organizations as mentioned earlier (where $B_{max} \leq 8$). In each case, the simplified loop nest L is executed for a total of n times, each with a different tile size. Let T_i be the execution time corresponding to the tile size S_i used. Given a tile size S_i, all parameters in T_{total} except α, β and γ are now known. We can find the values of α, β and γ by performing a linear curve fitting using the least-square method for T_{total} with respect to the n execution times, $T_1, \ldots, T_n$, obtained.

```
1  Compute the register usage per thread, R_T, using any tile size and thread organization.
2  for each tile size m = (m_1, ..., m_n) that satisfies the tile size constraint
3     Let S_TB (shared memory usage per block) be set as the shared memory usage per tile
4     for each t = (blockDim.x, blockDim.y, blockDim.z) that satisfies the blockDim constraint
5     Let R_TB = R_T × blockDim.x × blockDim.y × blockDim.z
6     Let B = min(B_W, B_R, B_S), where B_W, B_R and B_S are computed in Table 1.
7     Evaluate T_total given in (9) for the current tile size m and the current thread organization
      specified by gridDim = B × P and blockDim = t
8     if T_total < T_best // T_best is initialized to ∞
9        T_best = T_total
10       Record m as the best tile size so far (and set gridDim.x = B × P and blockdDim = t)
```

Fig. 5. An algorithm for automating tile size selection

3.4 Border Tiles

A border tile may execute faster than a full tile. If the i-th inter-tile wavefront that induces T^i_{t1} in (11) contains non-full border tiles, then T^i_{t1} may overapproximate the actual execution time of the wavefront. We can improve this inaccuracy with an estimate of $0.5 \times T_{\mathcal{TILE}}$ as the execution time of a border tile $\mathcal{TILE}$ by assuming that the average size of border tiles is half of a full tile.

4 Model-Driven Tile Size Selection

Given the estimated execution time of T_{total} in (9) for a tiled loop nest L as input, we employ an "educated" search to find automatically and efficiently an optimal tile size $m = (m_1, \ldots, m_n)$ for L and an associated thread organization, determined by `gridDim` and `blockDim`, used for realizing the optimal tiling. In this paper, a *tile layout* is determined by a tile size and a thread organization.

4.1 The Algorithm

We use two kinds of constraints to prune the search space:

Tile Size Constraint. The tile size, i.e., $L_{\mathcal{TILE}} = m_1 \times \cdots \times m_n$ is bounded from below by a data reuse rate $D = \frac{L_{\mathcal{TILE}}}{N_{mem}}$ (where N_{mem} is introduced in (6)) and from above by the size of shared memory. For DOACROSS loops, large tile sizes lead to higher data reuse rates. Thus, D must be larger than an empirical minimum threshold to ensure better intra-tile data locality.

`blockDim` Constraint. In NVIDIA GPUs, `blockDim.x` × `blockDim.y` × `blockDim.z` represents the number of threads per block. According to [16], the SP performance usually suffers with too many or too few threads. Furthermore, `blockDim.x` × `blockDim.y` × `blockDim.z` must be no smaller than the number of iterations contained in the largest intra-tile wavefront to ensure that every thread has some work to do. Thus, some small and large values of `blockDim.x` × `blockDim.y` × `blockDim.z` can be ignored.

Our algorithm for automating tile size selection for L is outlined in Figure 5. Recall that as shown in Figure 3, all tiles in a thread block are executed sequentially. Thus, for every type of resource listed in Table 1, the amount consumed

by a block is calculated on a per-tile basis. The basic idea is to perform an "educated" search when going through all tile layouts to find the one with the smallest execution time T_{total}. In line 1, the register usage per thread, denoted R_T, is measured independently of tile layouts used. This is because in each case the same code as shown in Figure 3 is compiled for each thread by NVIDIA's `nvcc` compiler. Finding R_T this ways speeds up the process for calculating R_{TB} in line 5. Similarly, in line 3, S_{TB} does not depend on `blockDim`. Once R_{TB} and S_{TB} are known, B_W, B_R and B_S are computed in line 6 as per Table 1.

4.2 The Framework

We have implemented our tile size selection technique using a combination of the Clan polyhedral representation extractor, Pluto's polyhedral parallel tiling infrastructure and CLooG code generator, as shown in Figure 6.

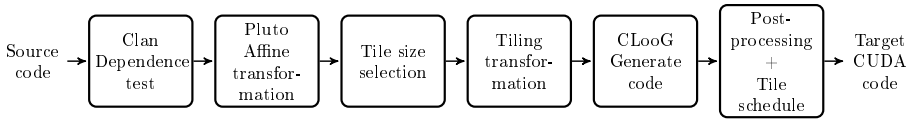

Fig. 6. A model-driven tile size selection framework

Our tile size selection module is invoked in the third step in the sequence.

5 Experiments

We use three representative DOACROSS kernels, 1D (3-point), 2D (5-point) and 3D (7-point) SOR solvers, to demonstrate the accuracy and efficiency of our tile size selection framework on an NVIDIA Tesla C1060 GPU (c.f. Table 1). Four problem sizes are discussed for each kernel, representing 12 different optimization problems for which best tile layouts (tile size/`blockDim` combinations) are solved.

Accuracy. It is impractical to measure the accuracy of our tile size selection framework for a kernel by comparing the actual execution time of the best tile layout found with the execution times of all tile layouts.

We have decided to evaluate this work empirically as is often done in automated performance tuning. For each of the 12 optimization problems discussed here, we have randomly sampled 1000 different tile layouts. The largest relative error (between the estimated execution time T_{total} and actual execution time) is observed to be within 6.05%. To see this graphically, the relative errors of 100 sampled tile layouts for each optimization problem are plotted in Figures 7 – 9. Let us look at the actual performance gap between the tile layout found by us and the best-performing one in each case. Let us consider a generic optimization problem O. Let T^m_{total} and R^m_{total} be the estimated and actual execution times of any tile layout m for O with the relative error being e_m. In particular, let T^{opt}_{total} and R^{opt}_{total} be the estimated and actual execution times of the the best tile layout *opt* predicted for O with the relative error being e_{opt}. The (worst)

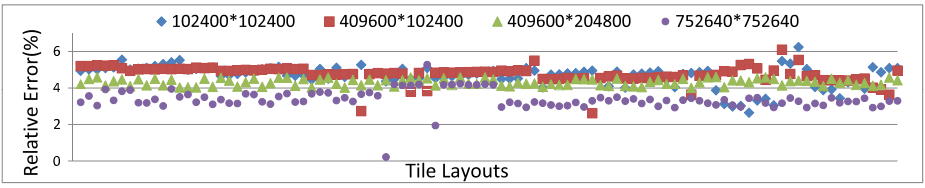

Fig. 7. 1D SOR: relative errors for 100 tile layouts in each of the four problem sizes

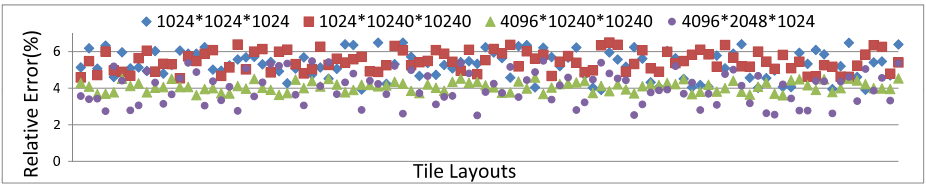

Fig. 8. 2D SOR: relative errors for 100 tile layouts in each of the four problem sizes

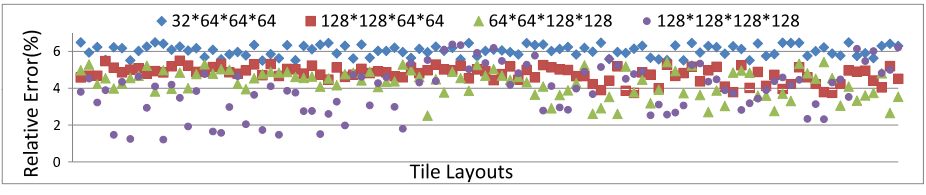

Fig. 9. 3D SOR: relative errors for 100 tile layouts in each of the four problem sizes

performance gap between *opt* and the best-performing one, m, is bounded by $(\frac{1+e_m}{1+e_{opt}} - 1) \times 100\%$, when $R_{opt} - R_m = T^{opt}_{total}/(1 + e_{opt}) - T^{m}_{total}/(1 + e_m)$, i.e., e_m is the largest, where $R_{opt} > R_m$. Based on our sampled tile layouts, the performance gaps are found to be 5.29%, 0.51%, 2.10% and 4.92% for the four problem sizes of 1D SOR (displayed from left to right in Figure 7), 4.33%, 1.31%, 5.14% and 2.01% for the four problem sizes of 2D SOR (Figure 8), and 0.66%, 2.28%, 6.05% and 1.30% for the four problem sizes of 3D SOR (Figure 9).

Search Time. Our algorithm is efficient in finding the best tile layout for a loop nest (on an Intel Xeon 2.0 GHz CPU). When tiling an n-dimensional loop nest that represents an $(n-1)$-D SOR solver with a tile size $m = (m_1, \ldots, m_n)$, m_1 represents the time dimension and $m_2, \ldots, m_n$ represent the $n-1$ spatial dimensions for the underlying mesh. Due to loop skewing, the worksets of different time slices of a tile are also skewed [10]. Thus, the data reuse rates of a tile for the 1D, 2D and 3D SOR solvers are expressed as a function of m and are bounded from above by m_2, $\frac{m_2 m_3}{m_2+m_3}$ and $\frac{m_2 m_3 m_4}{m_2 m_3 + m_2 m_4 + m_3 m_4}$, respectively, when $m_1 \to \infty$. For the 1D SOR solver, the data reuse rate induces a tile size constraint: $\frac{L_{TILE}}{N_{mem}} \geq 300$, where the threshold 300 is empirically set (Section 4.1). The search time is 238 secs over a search space of 3×10^6 tile layouts. For the 2D SOR solver, the tile size constraint is $\frac{L_{TILE}}{N_{mem}} \geq 6$. The search time is 369 secs

over a search space of 3.2×10^6 tile layouts. For 3D SOR, the data reuse rate imposes $\frac{L_{TTCE}}{N_{mem}} \geq 1$. The search time is 503 secs over a search space of 4.7×10^6 tile layouts.

References

1. Baghsorkhi, S.S., Delahaye, M., Patel, S.J., Gropp, W.D., Hwu, W.-m.W.: An adaptive performance modeling tool for GPU architectures. In: PPoPP 2010, pp. 105–114. ACM Press, New York (2010)
2. Baskaran, M.M., Ramanujam, J., Sadayappan, P.: Automatic C-to-CUDA Code Generation for Affine Programs. In: CC 2010, pp. 244–263 (2010)
3. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: PPoPP 2010, pp. 115–126 (2010)
4. Cui, H., Wang, L., Xue, J., Feng, X., Yang, Y.: Automatic library generation for blas3 on gpus. In: IPDPS 2011 (2011)
5. Cui, H., Xue, J., Wang, L., Yang, Y., Feng, X., Fan, D.: Extendable pattern-oriented optimization directives. In: CGO 2011, pp. 107–118 (2011)
6. Di, P., Wan, Q., Zhang, X., Wu, H., Xue, J.: Toward harnessing doacross parallelism for multi-gpgpus. In: ICPP 2010 (2010)
7. Fischer, S.: A parallel SSOR preconditioner for lattice QCD. Computer Physics Communications 98(1-2), 20–34 (1996)
8. Hackbusch, W.: Iterative solution of Large Sparse Systems of Equations. Applied Mathematical Sciences. Springer, Heidelberg (1993)
9. Hong, S., Kim, H.: An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In: ISCA 2009, p. 152 (June 2009)
10. Huang, Q., Xue, J., Vera, X.: Code tiling for improving the cache performance of PDE solvers. In: ICPP 2003, pp. 615–625 (2003)
11. Lee, S., Min, S.-J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: PPoPP 2009, pp. 101–110 (2009)
12. Liu, Y., Zhang, E.Z., Shen, X.: A Cross-Input Adaptive Framework for GPU Programs Optimization. In: IPDPS 2009, pp. 16–19 (2009)
13. Quarteroni, A., Valli, A.: Numerical Approximation of Partial Differential Equations. Springer, Heidelberg (1994)
14. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.-m.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: PPoPP 2008, pp. 73–82 (2008)
15. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: SC 2008, pp. 1–11 (2008)
16. Wong, H., Papadopoulou, M.M., Sadooghi-Alvandi, M., Moshovos, A.: Demystifying GPU microarchitecture through microbenchmarking. In: ISPASS 2010, pp. 235–246 (2010)
17. Xiao, S., Feng, W.-C.: Inter-block GPU communication via fast barrier synchronization. In: IPDPS 2010, pp. 1–12 (2010)
18. Xue, J.: Loop Tiling for Parallelism. Kluwer Academic Publishers, Dordrecht (2000)
19. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. In: PLDI 2010, p. 86 (May 2010)
20. Yuki, T., Renganarayanan, L., Rajopadhye, S., Anderson, C., Eichenberger, A.E., O'Brien, K.: Automatic creation of tile size selection models. In: CGO 2010, p. 190 (2010)