Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories

Muthu Manikandan Baskaran[1] Uday Bondhugula[1] Sriram Krishnamoorthy[1]
J. Ramanujam[2] Atanas Rountev[1] P. Sadayappan[1]

[1]Dept. of Computer Science and Engineering
The Ohio State University
2015 Neil Ave. Columbus, OH, USA
{baskaran,bondhugu,krishnsr,rountev,saday}@cse.ohio-state.edu

[2]Dept. of Electrical & Computer Engg. and
Center for Computation & Technology
Louisiana State University
jxr@ece.lsu.edu

Technical Report OSU-CISRC-2/08-TR05
February 2008

Abstract

Several parallel architectures such as GPUs and the Cell processor have fast explicitly managed on-chip memories, in addition to slow off-chip memory. They also have very high computational power with multiple levels of parallelism. A significant challenge in programming these architectures is to effectively exploit the parallelism available in the architecture and manage the fast memories to maximize performance.

In this paper we develop an approach to effective automatic data management for on-chip memories, including creation of buffers in on-chip (local) memories for holding portions of data accessed in a computational block, automatic determination of array access functions of local buffer references, and generation of code that moves data between slow off-chip memory and fast local memories. We also address the problem of mapping computation in regular programs to multi-level parallel architectures using a multi-level tiling approach, and study the impact of on-chip memory availability on the selection of tile sizes at various levels. Experimental results on a GPU demonstrate the effectiveness of the proposed approach.

Keywords Scratchpad memory, Data movement, Multi-level tiling, Graphics Processor Unit

1. Introduction and Motivation

Modern high-performance computer architectures have increasing numbers of processing elements on chip. Architects must ensure that memory bandwidth and latency are also optimized to exploit the full benefits of the available computational resources. Introducing a cache hierarchy has been the traditional way to alleviate the memory bottleneck. Caches are hardware-controlled, and it is difficult to model their exact behavior and to predict program execution times. While using caches, useful data may be evicted from the cache and replaced with other data without the programmer's control. Due to this and other reasons concerning performance and power, various modern parallel architectures have fast explicitly managed on-chip (local) memories, often referred to as *scratchpad memories*, in addition to slower off-chip (global) memory in the system. The scratchpad memories are software-managed and hence software has complete control over the movement of data into and out of such memories. The execution times of programs using scratchpad memories can be more accurately predicted and controlled. Scratchpad memories help to minimize memory load/store latency and maximize on-chip bandwidth by providing more paths without any issues about coherency.

Numerous challenges arise for a compiler writer provided with an architecture with explicitly managed memories. The compiler has to make good decisions on what elements to move in and move out of local memory, when to move them, and how to efficiently access the elements brought into local memory, while ensuring program correctness. Programmers using architectures with scratchpad memories shoulder the burden of orchestrating the movement of data between global and local memory.

Another significant problem to be addressed in modern high-performance multi-level parallel architectures is the exploitation of available parallelism. Parallelization of arbitrary regular programs in these architectures is challenging as the architecture imposes a number of constraints that have to be addressed in order to effectively map the computation onto the parallel units of the architecture. The computation mapping should utilize the benefits of local memories and hence the mapping is constrained by the amount of local memory available.

In this paper we develop approaches to address the challenges of effective automatic data management in scratchpad memories and of effective mapping of computation in regular programs to architectures with multiple levels of parallelism. To address the first problem, we develop a framework that automatically creates buffers in local memory for storing portions of data that are accessed in a block of computation, and determines array access functions for local buffer references. The framework also automatically generates code to move data between local memory and global memory. To address the second problem, we perform multi-level tiling to distribute computation on multiple levels of parallel units in the architecture.

The rest of the paper is organized as follows. Section 2 introduces the polyhedral model for representing programs and transformations. Section 3 presents the framework to perform automatic data allocation and code generation for data movement in scratchpad memories. Section 4 describes the multi-level tiling approach for computation mapping on multi-level parallel architectures. Section 5 gives an overview of the GPU architecture that is used as a testbed for the experiments presented in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2. Overview of Polyhedral Model

This section provides some background information on the polytope/polyhedral model, a powerful algebraic framework for representing programs and transformations [29, 33]. The polyhedral model is used by our

framework to perform automatic allocation and data movement in scratchpad memories (discussed in detail in Section 3).

A hyperplane is an $n-1$ dimensional affine subspace of an n-dimensional space; a hyperplane can be represented by an affine equality. A halfspace consists of all points of an n-dimensional space that lie on one side of a hyperplane (including the hyperplane); it can be represented by an affine inequality. A polyhedron is the intersection of finitely many halfspaces. A polytope is a bounded polyhedron. A polytope is represented as

$$M\mathbf{x}+\mathbf{b} \geq \vec{0}$$

where x is a vector of variables, b is a constant vector, and M is a matrix in which every row describes a hyperplane that bounds the polytope.

In the polyhedral model, a statement s surrounded by m loops is represented by an m-dimensional polytope, referred to as an iteration space polytope. The coordinates of a point in the polytope (referred to as the iteration vector $\vec{i_s}$) correspond to the values of the loop indices of the surrounding loops, starting from the outermost one. In this work, we focus on regular programs where loop bounds are affine functions of outer loop indices and global parameters (e.g., problem sizes) and similarly, array access functions are also affine functions of loop indices and global parameters. Hence the iteration space polytope I_s can be defined by a system of affine inequalities derived from the bounds of the loops surrounding s. Using matrix representation in *homogeneous* form to express systems of affine inequalities, the iteration space polytope can be represented as

$$I_s.\begin{pmatrix} \vec{i_s} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0}$$

where I_s is a matrix representing loop bound constraints and $\vec{p}$ is a vector of global parameters. Each point of the polytope corresponds to an instance of statement s in program execution.

Affine array access functions can also be represented using matrices. If $a[\mathcal{F}_{ras}(\vec{i_s})]$ is the rth reference to an array a in statement s with a corresponding iteration vector $\vec{i_s}$, then

$$\mathcal{F}_{ras}(\vec{i_s}) = F_{ras}.\begin{pmatrix} \vec{i_s} \\ \vec{p} \\ 1 \end{pmatrix}$$

where F_{ras} is a matrix representing an affine mapping from the iteration space of statement s to the data space of array a. Each row in the matrix defines a mapping corresponding to a dimension of the data space. Hence the image of an iteration space polytope I_s by an affine access function F_{ras} defines the set of elements accessed by the affine reference, i.e., the data space accessed by the affine reference. In the subsequent discussion, we denote the image of I_s by F_{ras} as $F_{ras}I_s$.

There has been a significant body of work on dependence analysis in the polyhedral model [15, 34, 39]. We now discuss briefly the representation of dependences in the polyhedral model. An instance of statement s (denoted by iteration vector $\vec{i_s}$) depends on an instance of statement t (denoted by iteration vector $\vec{i_t}$) if $\vec{i_s}$ and $\vec{i_t}$ are valid points in the corresponding iteration space polytopes, they access the same memory location,

and $\vec{i_s}$ is executed before $\vec{i_t}$. Since array accesses are assumed to be affine functions of loop indices and global parameters, the constraint that defines conflicting accesses of memory locations can be represented by an affine equality (obtained by equating the array access functions in source and target statement instances). Hence all constraints to capture a dependence can be represented as a system of affine inequalities with a corresponding polyhedron (referred to as a *dependence polyhedron*).

The technique of employing the polyhedral model to find (affine) program transformations has been widely used for improvement of sequential programs (source-to-source transformation) [16, 17] as well as automatic parallelization of programs [28, 21, 18, 11]. An affine transform of a statement s is defined as an affine mapping that maps an instance of s in the original program to an instance in the transformed program. The affine mapping function of a statement s is

$$\phi_s(\vec{i_s}) = \mathcal{C}_s . \begin{pmatrix} \vec{i_s} \\ \vec{p} \\ 1 \end{pmatrix}$$

When $\mathcal{C}_s$ is a row vector, the affine mapping ϕ_s is a one-dimensional mapping. An m-dimensional mapping can be represented as a combination of m (linearly independent) one-dimensional mappings, in which case $\mathcal{C}_s$ is a matrix with m rows.

3. Automatic Data Management in Scratchpad Memories

In this section we discuss in detail the proposed framework (based on the polyhedral model discussed in Section 2) for effective automatic data management in scratchpad memories. We discuss three different aspects of the framework: (1) automatic allocation of storage space (in the form of arrays) in scratchpad memories for holding portions of data accessed in a block of a program, (2) determination of access functions of references to arrays in scratchpad memories, and (3) automatic generation of code for moving data between scratchpad (local) memory and off-chip (global) memory. The local memory storage allocation is done independently of whether the access functions of various array references are uniformly generated or non-uniformly generated. We create local memory storage for each non-overlapping region of the data space of an array that is accessed in a program block. By default, data is moved in to local memory at the start of the block and moved out after the completion of the block. Data dependences are not violated because of the framework's automatic storage allocation and data movement.

In modern parallel architectures such as the Cell processor, any data that is accessed in a computation has to be moved into scratchpad memory before access, as data cannot be accessed from global memory during computation. But in architectures such as GPUs, data can be accessed from both global memory and scratchpad memory during computation. For such architectures, the framework optimally moves only data that have sufficient reuse.

3.1 Details of the Framework

The framework takes as input the iteration spaces of all statements in a program block as well as the access functions of all array references. The procedure explained below is applied to all arrays (one at a time) in the block.

For an array A, let $S_1, S_2, \ldots, S_q$ be the statements in the given program block containing references to A, and let I_k represent the iteration space of statement S_k ($1 \leq k \leq q$). Let $F_k^1, F_k^2, \ldots, F_k^p$ be the matrices representing

Algorithm 1 Reuse benefit calculation algorithm

Input Set of data spaces (D), Number of references in D (N_{ref}), Iteration space dimensionality of each reference ($IS_i, 1 \leq i \leq N_{ref}$), Rank of affine function of each reference ($R_i, 1 \leq i \leq N_{ref}$)

```
for i=1 to N_ref do
    if R_i < IS_i then
        Mark yes
    end if
end for
if Not Marked yes then
    if Volume(overlapped regions in D) > δ× Volume(D) then
        Mark yes
    end if
end if
```

Output Yes, if beneficial reuse, No, otherwise

the affine read reference functions of A in statement S_k, and let G_k be the matrix representing the affine write reference function of A (if there is a write reference of A in S_k).

As discussed in Section 2, the data space accessed by a reference (represented by an access function matrix F) in a statement represented by an iteration space I is given by FI. Hence the set of all data spaces accessed by all read references of A in all statements in the program block is

$$\mathcal{DS}_r^A = \{F_k^l I_k \ \mid \ 1 \leq l \leq p \ \ \wedge \ \ 1 \leq k \leq q\}$$

Similarly, the set of all data spaces accessed by all write references of A in the block is

$$\mathcal{DS}_w^A = \{G_k I_k \ \mid \ 1 \leq k \leq q\}$$

The set of all data spaces accessed by all references of A in the block is

$$\mathcal{DS}_{rw}^A = \mathcal{DS}_r^A \cup \mathcal{DS}_w^A$$

We partition the set of all data spaces $\mathcal{DS}_{rw}^A$ into maximal disjoint sets $\mathcal{DS}_{rw_{d_1}}^A, \mathcal{DS}_{rw_{d_2}}^A, \ldots, \mathcal{DS}_{rw_{d_m}}^A$ such that each partition has a subset of data spaces each of which is non-overlapping with any data space in other partitions, i.e.,

$$\forall d \in \mathcal{DS}_{rw_{d_i}}^A, \forall e \in \mathcal{DS}_{rw_{d_j}}^A \ \text{ s.t. } \ 1 \leq i, j \leq m \wedge j \neq i: \ \ d \cap e = \emptyset$$

The problem of partitioning is solved by mapping it to an equivalent problem of finding connected components of an undirected graph. The undirected graph is created with vertices representing each data space in the set $\mathcal{DS}_{rw}^A$; an edge exists between two vertices if the intersection of the data spaces corresponding to the vertices is not empty.

The convex union or the minimum convex polytope that encloses all data spaces in a partition $\mathcal{DS}_{rw_{d_i}}^A$ is

$$\mathcal{CDS}_{rw_{d_i}}^A = \text{ConvexHull}(\mathcal{DS}_{rw_{d_i}}^A)$$

3.1.1 Determining Local Memory Storage

We now describe the procedure that automatically allocates storage in local memory for portions of array A that are accessed in the given program block. We generate one local memory array for each partition $\mathcal{DS}^{A}_{rw_{d_i}}$.

Algorithm 1 explains a procedure to determine if a partition of data spaces has sufficient reuse in a program block. When the rank of the access matrix of an array reference is less than the iteration space dimensionality of the statement in which it is accessed, the data elements of the array accessed in the reference are said to have an order of magnitude (or non-constant) reuse. Thus, the condition for non-constant reuse of a data space of an array a accessed in a reference $\mathcal{F}_{ras}(\vec{i_s})$ is

$$rank(F_{ras}) < dim(\vec{i_s}) \tag{1}$$

If a given partition of data spaces has at least one reference whose accessed data space has non-constant reuse (i.e., it satisfies Condition (1)), then the partition is marked as beneficial to be copied to scratchpad memory. Otherwise we check if the data spaces in the partition have significant constant reuse. Constant reuse in the set is estimated by considering each pair of data spaces, determining the volume of their intersection, and summing up these volumes. If the sum of the volumes constitutes a significant portion, determined by a fraction δ, of the total volume of the set of all data spaces, then the partition is marked as beneficial. We have empirically fixed a value of 30% for δ.

The procedure explained below is applied to each partition of data spaces $\mathcal{DS}^{A}_{rw_{d_i}}$ to determine local memory storage space for the partition. For architectures such as GPUs, only those partitions that are marked as beneficial by Algorithm 1 are considered. The overall approach is summarized in Algorithm 2.

For each partition of data spaces $\mathcal{DS}^{A}_{rw_{d_i}}$ we consider the corresponding convex hull $\mathcal{CDS}^{A}_{rw_{d_i}}$ and find the lower and upper bounds of each dimension of this convex hull. The bounds are determined in the form of an affine function of parameters of the program block, using the Parametric Integer Programming (PIP) software [14]. These bounds determine the size of the local memory array created for the partition. Let the dimensionality of the convex hull be n. Let $i_1, i_2, \ldots, i_n$ represent the variables denoting each dimension of the convex hull and let lb_k and ub_k be the lower and upper bounds of dimension variable i_k. Local memory storage created for a partition of accessed data spaces of array A is an array of dimension n with size $(ub_1 - lb_1 + 1) \times (ub_2 - lb_2 + 1) \times \cdots \times (ub_n - lb_n + 1)$. The order of array dimensions of the local memory array follow that of the global memory array.

3.1.2 Determining Access Functions of Local Memory Array References

For each reference to the original array A in the given program block, we find the corresponding access function for the local memory array reference. The dimensionality (say m) of the original global memory array A might be greater than the dimensionality (say n) of the local memory array created for a partition of accessed data spaces of A. We use CLooG [10], an efficient code generator tool, to find the bound expressions of each dimension of the convex hull of the partition of accessed data spaces. The bound expressions generated from CLooG are represented as a loop structure with n nested loops. The dimensions of the original data space that do not appear in the convex union polytope (convex hull) are represented as affine functions of dimensions that appear in the polytope, and program parameters, in their respective positions within the loop structure.

Algorithm 2 Automatic data allocation algorithm

Input Iteration space polyhedra, Affine array access functions

1: **for** each array A **do**
2: **for** each reference of the array **do**
3: Find the data space accessed by the reference
4: **end for**
5: Partition the set of all data spaces into maximal disjoint sets such that each partition has a subset of data spaces each of which is non-overlapping with any data space in other partitions
6: **for** each partition of data spaces **do**
7: Find the convex union of data spaces in the partition
8: Find the lower and upper bounds of each dimension of the convex union, as an affine function of program parameters. Let the number of dimensions be n. Let the dimension variables be $i_1, i_2, \ldots, i_n$. Let lb_k and ub_k be the lower and upper bounds of the dimension variable i_k.
9: Define the local storage for a partition of accessed data spaces of array A as an array of dimension n and size $(ub_1 - lb_1 + 1) \times (ub_2 - lb_2 + 1) \times \cdots \times (ub_n - lb_n + 1)$
10: **end for**
11: **end for**

Output Local memory storage for each non-overlapping accessed region of original arrays

For any array access $\mathcal{F}(\vec{y})$, each row of the access function matrix F represents the array subscript of a dimension in the original data space. Let F' be the matrix that is derived from F by removing the rows (if any) corresponding to the dimensions in the original array that do not appear in the local memory array. (Note that $F' = F$ when $m = n$).

For any reference $A[\mathcal{F}(\vec{y})]$ in the original program block, the corresponding local memory array access function is

$$\mathcal{F}'(\vec{y}) - g, \quad \text{where} \quad g = (lb_1, lb_2, \ldots, lb_n)^T$$

3.1.3 Generating Data Movement Code

This subsection describes the procedure to generate code for data movement. This procedure is applied to each partition of data spaces $\mathcal{DS}^A_{rw_{d_i}}$ for which a local memory array is created.

From $\mathcal{DS}^A_{rw_{d_i}}$, we select the data spaces that are accessed by read references. We generate the loop structure of the code that moves data from global memory to local memory by scanning the selected data spaces using CLooG. Similarly, from $\mathcal{DS}^A_{rw_{d_i}}$, we select the data spaces that are accessed by write references and use CLooG to generate the loop structure of the code that moves data from local memory to global memory by scanning the selected data spaces. CLooG scans the data spaces in an efficient way such that the generated loop structure leads to single load/store of each data element that is read/written even if the accessed data spaces of references are overlapping.

Having generated the loop structures, the loop body of the data move in and move out code is generated as follows. Let the dimensionality of the original array and the local memory array be m and n, respectively. Let H be a $m \times n$ matrix derived from an $n \times n$ identity matrix Id_n by adding a row (in the respective position) for each of the $m - n$ dimensions that do not appear in the convex union polytope $\mathcal{CDS}^A_{rw_{d_i}}$. The added row represents the corresponding dimension as affine function of dimensions that appear in the polytope, and program parameters.

(Note that $H = Id_n$ when $m = n$). Let $\vec{y}$ be the iteration vector of the loop structure of data move in (and data move out) code.

The loop body of the data move in code for each local array L_i created for A is

$$L_i[Id_n.\vec{y} - g] = A[H.\vec{y}]$$

and the loop body of the data move out code is

$$A[H.\vec{y}] = L_i[Id_n.\vec{y} - g]$$

$$\text{where} \quad g = (lb_1, lb_2, \ldots, lb_n)^T$$

To estimate an upper bound on the volume of data moved in to the local memory array created for $\mathcal{DS}^A_{rw_{d_i}}$, we partition the set of data spaces in $\mathcal{DS}^A_{rw_{d_i}}$, that are accessed by read references, into maximal non-overlapping subsets of data spaces (as explained earlier in the section), and find the space required in the local memory array for each such partition (using the procedure explained in Algorithm 2). The upper bound on the volume of data moved in to the local array is given by the total space needed in the local array for all such partitions. Similarly, we estimate the upper bound on the volume of data moved out of the local memory array by finding the total space needed for all non-overlapping partitions of data spaces in $\mathcal{DS}^A_{rw_{d_i}}$, that are accessed by write references.

We use the Polylib [32] tool to perform operations over polytopes, such as finding the image of iteration space polytopes formed by affine functions, finding the union of data spaces, and finding the convex union of data spaces.

Figure 1 shows an example that illustrates automatic data allocation in local memory storage and code generation for data movement.

3.1.4 Optimizing Data Movement

All data that are accessed by read references in a program block need not be moved in to scratchpad memory; similarly, all data that are accessed by write references need not be written out from scratchpad memory. Only those data elements that are read by statement instances in the block but whose values are written by earlier statement instances outside the block, need to be moved in. Of course, data elements corresponding to an input array (array that is only read in the program but not written) also need to be brought in. Similarly, data elements that are written by statement instances in the block but not read by any statement instance outside the block need not be copied out to global memory from scratchpad memory, unless the data elements belong to an output array (array that is only written in the program but not read).

The optimal strategy for determining data elements that need to be copied in and copied out requires data dependence information. We are given the iteration spaces of all statements in a program block and the set of all dependence polyhedra. For each true dependence (defined by a polyhedron), we identify each statement instance in the block that is a target of the dependence but whose corresponding source statement instance does not belong to the block. The data accessed due to the array read references involved in true dependences in the set of all identified statement instances constitute the data that needs to be copied into local memory (in addition to data belonging to input arrays accessed in the block) for the computation in the block. Similarly, for each true dependence, we identify each statement instance in the block that is a source of the dependence but whose

corresponding target statement instance does not belong to the block. The data accessed due to the array write references involved in true dependences in the set of all identified statement instances constitute the data that needs to be copied out from local memory (in addition to data belonging to output arrays written in the block) after the computation in the block.

The current implementation of the framework takes iteration spaces of statements and affine array access functions of references as input, and creates local memory arrays and data movement code. In future work we plan to implement the optimization outlined above, based on data dependence information.

4. Tiling for Multiple Levels of Parallelism

In this section we present in detail the approach of mapping computation in an affine program to different levels of parallel units in a multi-level parallel architecture that has explicitly managed on-chip memories. We use a multi-level tiling approach that addresses the constraints imposed on tile sizes by the availability of on-chip memory.

4.1 Details of the Approach

The architecture we consider for further explanation has the following components: (1) a slow global memory, (2) a set of parallel units at an outer level that communicate with each other through the global memory space, (3) a set of parallel units within each outer-level parallel unit, and (4) a local fast explicitly managed scratchpad memory within each outer-level parallel unit shared by the inner-level parallel units. The number of software parallel processes or threads executed in the system can be higher than the number of parallel processors in the hardware. Each outer-level parallel process can be thought of as logical grouping of a set of inner-level parallel processes. When more than one process is launched on an outer-level parallel unit, these processes divide the scratchpad memory among themselves. In this case, the inner-level processes that are part of one outer-level process have available to them only a portion of the local scratchpad memory of the outer-level unit; these inner-level processes share this portion among themselves.

Given any input program, the first step is to find the parallelism available in the computation. Our approach uses the framework developed by Bondhugula et al. [7] for this purpose. Given any affine input program, the framework finds an optimal set of affine transformations (or, equivalently, tiling hyperplanes) for each statement to minimize the volume of communication between tiles and also to improve data reuse in each tile. The framework finds bands of permutable loops that can be tiled and also identifies points in execution where synchronization is required. A band can have a single sequential loop, or it can have multiple loops found in the increasing order of communication volume induced due to the loop. In our approach, we consider the outermost band that has multiple permutable loops, and treat the communication-free loops in the band, if any, as space loops. If there are no communication-free loops in the outermost band, we treat all but the last loop as space loops, in order to achieve pipeline parallelism. Having identified the bands of permutable loops and the space and time loops, we proceed to perform multi-level tiling of the space loops to distribute the available parallelism across the various levels of parallel units in the system. By default, we do as many levels of tiling as the number of levels of parallel units. But due to constraints imposed by memory availability at different levels (as explained later), we perform additional levels of tiling when needed. For these additional levels of tiling, which are done within tiles that are distributed across parallel processes, we tile all permutable loops.

In the two-level parallel architecture considered, for programs that require synchronization across outer-level parallel processes, all processes involved in synchronization need to be launched and active on the outer-level hardware parallel processors. In such cases, for a specific total number of outer-level parallel processes that are launched on the same outer-level processor, there is an upper limit on the amount of local memory that is available to each process. This limit is given by the total capacity of local memory in the processor, divided by the number of processes assigned to this processor. However, for programs that do not require synchronization across outer-level parallel processes, all these processes need not be active at the same time on the outer-level hardware parallel processors. This allows some of them to be launched after the completion of others, which could increase the amount of available local memory per process. In such cases, the upper limit on the amount of local memory that is available to each process, that is launched on an outer-level processor, is ideally the total capacity of local memory in the processor.

The procedure to perform multi-level tiling for the two-level parallel architecture considered, is as follows. We fix the number of parallel processes at outer and inner levels to be a multiple of the number of physical parallel processors at the level. We first perform an outer level of tiling of the space loops that equally distributes tiles across outer-level parallel processes. If the tile in an outer-level process is large enough such that it requires more local memory than the available amount, it becomes necessary to introduce one more level of tiling, in order to limit the amount of needed local memory. In this case we split the tile in an outer-level parallel process into sub-tiles (that are executed sequentially within the outer-level tile) such that each sub-tile requires an amount of local memory that is no higher than the fixed upper limit. We find an optimal set of tile sizes that defines an atomic unit of computation in an outer-level tile under the constraint of limited local memory availability, using the algorithm described in Section 4.3. Once the outer level tiling is done, we perform an inner level of tiling of the space loops that equally distributes the iterations of an atomic unit of computation executed in an outer-level parallel process among the inner-level parallel processes.

Figure 2 and Figure 3 illustrate an example in which multiple levels of tiling are done to exploit various levels of parallelism available in the system.

4.2 Optimal Placement of Data Movement Code

With the tiling hyperplanes (that determine the shape of a tile executed atomically in an outer-level parallel process) known, the iteration spaces of statements in a tile, parameterized by tile sizes, are determined. The iteration spaces of statements in a tile, along with the affine array access functions for each reference in the tile are given as input to the framework described in Section 3 to determine the local memory storage needed for data accessed in the tile (as a function of tile sizes) and to generate code to move data between local memory and global memory. By default, code to move data in to local memory allocated for a tile is placed in the program structure at the beginning of execution of the tile (computational block) and code to move modified data out to global memory is placed at the end of execution of the tile. The tiling loops form a loop nest over the data movement code and tile code.

When an array access function does not depend on a loop iterator in the iteration space, then the loop is a *redundant loop* for the reference. If all references of a local memory array have one or more common redundant loops in the loop nest of tiling loops, then the data movement code of the array is hoisted and placed at a level in the loop nest of tiling loops such that any tiling loop that is below the level in the loop nest is a redundant loop

identified for the array. By doing such hoisting, the data in the local memory array is reused, if possible, across various computational blocks. Placing data movement code outside redundant loops helps in the selection of optimal tile sizes (for tiles that are sequentially executed in an outer-level parallel process).

4.3 Tile Size Search Algorithm

The goal of this algorithm is to find optimal sizes for tiles that determine the atomic units of computation executed in an outer-level parallel process, under the constraint that the active local memory used by the process does not exceed a given upper limit M_{up}. The algorithm tries to minimize data movement cost between local memory and global memory.

The data movement cost is directly affected by the volume of data that is being moved and the number of occurrences of data movement. The movement is done in parallel by the inner-level processes and there has to be a synchronization among these processes every time the data is moved. Hence the data movement cost, C, is modeled as

$$C = N \times ((P \times S) + \frac{V \times L}{P})$$

where N is the number of occurrences of data movement, P is the number of inner-level processes, S is the synchronization cost per process per occurrence of data movement, V is the volume of data moved each time, and L is the cost of transferring a data element. There is usually an architecture-dependent constraint on the minimum number of inner-level processes to be chosen, below which the resources are under-utilized and overlap of computation and load/store into local memory is very poor. Let this lower limit be P_{low} and $P \geq P_{low}$.

Consider tiling of a loop nest with maximum depth m. Let the index range of the ith loop in the nest be N_i and let t_i be the optimal tile size to be found for the ith loop in the nest. Let the number of local memory arrays created in a tile be nl and r_j be the optimal position in the loop nest where data movement code of jth local memory array is placed. Using Algorithm 2, we determine the size of each local memory array created in a tile, as a function of the tile sizes. Let $M_1, M_2, \ldots, M_{nl}$ represent the size of each local memory array. Let V_j^{in} and V_j^{out} be the upper bounds on the volume of data moved in to and moved out of jth local memory array, respectively, determined as a function of tile sizes, as explained in Section 3.1.3. Let I denote the set of arrays to which data is moved in and O denote the set of arrays from which data is moved out. The tile size search algorithm is phrased as an optimization problem that minimizes the cost of data movement between local memory and global memory. The constraints ensure that (1) all tile sizes are greater than zero but lesser than or equal to the corresponding loop index range, (2) total memory required by a tile is within the given upper limit M_{up}, and (3) tile sizes are large enough to keep all inner-level processes busy.

The optimization problem is formulated below.

Variables:

$$t_1, t_2, \ldots, t_m$$

Constraints:

$$\forall i : 1 \leq i \leq m, t_i > 0$$

$$\forall i : 1 \leq i \leq m, t_i \leq N_i$$

$$\sum_{i=1}^{nl} M_i \leq M_{up}$$

$$t_1 \times t_2 \times \cdots \times t_m \geq P$$

Objective function:

$$\text{minimize} \quad \sum_{k \in I} (\prod_{i=1}^{r_k} \frac{N_i}{t_i} \times ((P \times S) + \frac{V_k^{in} \times L}{P})) +$$

$$\sum_{k \in O} (\prod_{i=1}^{r_k} \frac{N_i}{t_i} \times ((P \times S) + \frac{V_k^{out} \times L}{P}))$$

The above optimization problem is a nonlinear constrained optimization problem that can be solved by a technique such as sequential quadratic programming (SQP). Note that we need to relax the integer constraints on $(t_1, t_2, \ldots, t_m)$ to lie in $\mathbf{R}^m$ instead of $\mathbf{Z}^m$, solve the optimization problem, then round off the result to the closest integral vector. A detailed discussion on SQP is presented in [4].

5. Overview of GPU Architecture

GPU architectures are representative of the class of multi-level parallel architectures with explicitly managed memories. We have used GPUs as the experimental testbed architecture to implement and illustrate the benefits of our work discussed in Section 3 and Section 4. This section presents a brief overview of the GPU architecture.

The GPU architecture has a set of multiprocessors (MIMD units) and within each multiprocessor has a set of processor cores executing in a SIMD fashion (referred to as SIMD units in further discussion). Hence the architecture exhibits a two-level parallelism, namely, parallelism across the multiprocessors (MIMD units) and parallelism across the various SIMD units within a multiprocessor. The multiprocessors communicate through an off-chip DRAM memory, which has very high access latency. The SIMD units within a multiprocessor communicate through a fast local scratchpad memory (referred to as shared memory) that resides in the multiprocessor. Programming GPUs for general-purpose applications is enabled through the Compute Unified Device Architecture (CUDA) programming model [30]. The CUDA programming model abstracts the multiprocessors as a grid of virtual processors called thread blocks, and abstracts the SIMD units within a multiprocessor as a grid of virtual processors called threads. The execution model of a GPU device is such that a grid of thread blocks is executed by running one or more blocks on each MIMD unit, and each block is split into SIMD groups of threads called warps. The size of a warp in NVIDIA GeForce 8800 GTX is 32. Hence the lower limit on the number of threads used, P_{low} (as defined in Section 4.3), is fixed at 32 for experiments on NVIDIA GeForce 8800 GTX.

Multi-level tiling, as described in Section 4, is performed at an outer level across the thread blocks mapped to the MIMD units, and at an inner level across the threads mapped to the SIMD units. In general, given a set of parallel or space loops for a program, the space loops are tiled and the tiles are allocated/scheduled to execute on the thread blocks. The parallel iterations within a tile are executed in a SIMD fashion across the threads of a thread block. Based on the availability of shared memory, the tile executing in a thread block is divided into sub-tiles that are executed sequentially in a thread block.

In GPUs the number of concurrent thread blocks at any point of time is determined by the amount of shared memory that is needed by any thread block. If the shared memory required by a thread block is M bytes and

if the total available shared memory in the device is X bytes, then the maximum number of concurrent thread blocks in the device cannot exceed $\frac{X}{M}$. The shared memory that is available in a MIMD unit in a NVIDIA GeForce 8800 GTX is 16 KB and there are 16 MIMD units; hence, the maximum number of concurrent thread blocks cannot exceed $\frac{2^{18}}{M}$.

6. Experimental Results

The experiments were conducted on a NVIDIA GeForce 8800 GTX GPU device. The device has 768 MB of DRAM and has 16 multiprocessors (MIMD units) at 675 MHz. Each multiprocessor has 8 SIMD units running at twice the clock frequency of the multiprocessor and has 16 KB of scratchpad memory (referred to as shared memory in the context of CUDA). The CUDA kernels are compiled using the NVIDIA CUDA Compiler (nvcc) to generate the device code that is launched from the CPU (host). The CPU is an Intel Core2 Duo processor at 2.13 GHz with 2 MB L2 cache. The GPU device is connected to the CPU through a 16-x PCI Express bus. The host programs are compiled using the gcc compiler with -O3 optimization flag.

We broadly conducted our experimental study on two kernels, namely Mpeg4 Motion Estimation (ME) kernel and 1-D Jacobi kernel, the former requiring no synchronization across thread blocks and the later requiring synchronization across thread blocks. For efficient execution of the kernels, we performed multi-level tiling on the kernels and effectively managed data in scratchpad memory using our automatic data management framework.

We first present the experimental results that show the benefits of using scratchpad memory to efficiently access data for computation. Fig. 4 and Fig. 5 illustrate the benefits of efficient data access using scratchpad memory and also exemplify the high speedup achieved in running the kernel in GPU in contrast to running in CPU. The speedup of the implementation utilizing scratchpad memory is 8x for Mpeg4 ME kernel and 10x for Jacobi kernel over that using only GPU DRAM. The speedup over CPU performance is over 100x for Mpeg4 ME kernel and 15x for Jacobi kernel.

The original and parallel tiled code structure of the Mpeg4 ME kernel are shown in Fig. 2 and Fig. 3. For various problem sizes, we conducted experiments to analyze the performance of the Mpeg4 ME kernel. The results are shown in Fig. 6. The number of thread blocks was chosen as 32 and the number of threads as 256. The i and j loops in Fig. 2 are the space loops that are tiled across thread blocks and threads, and k and l loops are the time loops. The sizes of tiles that distribute computation across thread blocks are set by dividing the problem size equally (except for boundary) among the thread blocks. The time loops k and l in this kernel iterate over very small index range (16 in our experiment) and hence the data accessed in the iteration space of the loops do not occupy much space in the scratchpad memory. The tile search algorithm described in Section 4.3 found tile sizes of 32, 16, 16 and 16 for i, j, k and l loops to be optimal as these tile sizes lead to lesser data movement cost by reducing the number of data movements, given the scratchpad memory constraint. From the results shown in Fig. 6, it is clear that the tile sizes chosen by the algorithm gave better performance than other tile sizes for various problem sizes.

For the implementation of Jacobi-1D kernel that has a space loop surrounded by a time loop, we used the framework discussed in [27] to modify the tiled code to enable concurrent start of execution in all processes, and performed multi-level tiling over the modified code. We ran experiments on the kernel for various problem sizes that can completely fit in the total scratchpad memory available in the GPU device and obtained results

that are depicted in Fig. 7. The number of time iterations was chosen as 4096 and time tile size as 32. The problem size was equally divided (except for boundary) by the number of thread blocks used, to set the size of space tile executed per thread block. The number of threads used was 64. For problem sizes that completely fit in the total scratchpad memory available in the device, the number of thread blocks has no constraint. However, since the kernel requires synchronization across all thread blocks, for very high number of thread blocks, the computation done by a thread block is too small to hide the synchronization cost incurred. The same behavior is illustrated by Fig. 7. Performance enhances as the number of thread blocks increases, till a point, and then decreases when the synchronization cost dominates over the amount of computation done in a thread block.

For larger problem sizes that have to do be tiled to fit in the scratchpad memory, we fixed the number of thread blocks to be 128, fixed empirically from the experimental results shown in Fig. 7, to allow better concurrency and incur less synchronization cost. The active scratchpad memory used by a thread block was hence limited to 2^{11} bytes or 2^9 words. The number of threads used was fixed at 64. The tile search algorithm described in Section 4.3 gave a space tile size of 256 and time tile size of 32 to be optimal for minimizing the data movement cost between scratchpad memory and global memory. The experiments confirmed the same, as indicated by the results in Fig. 8 that show the performance of the kernel for various tile sizes for different problem sizes.

7. Related Work

In this section we discuss prior work that has addressed compiler issues in multi-level parallel architectures and architectures with explicitly managed scratchpad memories.

Schreiber and Cronquist [38] have proposed an approach to do near-optimal allocation of scratchpad memories and near-optimal reindexing of array elements in scratchpad memories. Their approach generates separate storage efficient local arrays for each equivalent class of uniformly generated references. Hence if data accessed due to two references, belonging to different classes, are overlapping, then two different local arrays would be created to hold the overlapping accessed data spaces. Anantharaman and Pande [1] perform data partitioning on arrays into portions to be kept in local memory and global memory. They compute a bounding box for each equivalent group of uniformly generated references as in the case of [38]. Eisenbeis et al. [12] consider elements to move to local memory from a view of individual iteration of a loop nest instead of an atomic unit of computation of the program. Kandemir et al. [25] propose an approach for dynamically managing scratchpad memories, but they handle only uniformly generated affine references.

The idea of estimation of the number of references to an array in order to predict cache effectiveness has been discussed by Ferrante et al. [19] and Gallivan et al. [20]. The idea of finding image of the iteration space onto the array space to optimize global transfers has been discussed in [20]; but only a framework for estimating bounds for the number of elements accessed was given. Ferrante et al. gave exact values for uniformly generated references but did not consider multiple references. Also, for non-uniformly generated references, arbitrary correction factors were given for arriving at lower and upper bounds for the number of distinct references. Clauss[9] and Pugh [35] have presented more expensive but exact techniques to count the number of distinct accesses.

There has been a significant amount of research on memory reduction and optimization of data locality for embedded single-processor-on-chip (SOC) systems. In the case of memory optimizations, Panda et al., Balasa et al., and the IMEC group have derived several transformations for improving memory performance

on embedded systems [3, 8, 31, 36, 40]. Their work is a collection of techniques that form a custom memory management methodology referred to as *data transfer and storage exploration* (DTSE). There is a large body of work on estimating the memory requirements of loops [3, 8, 36, 40] (and references therein). Most of these works assume the given sequential execution order and find the memory requirements. A number of works have addressed scratchpad memory management [22, 24, 25] (to name a few).

Multi-level tiling approach has been employed in various contexts such as tiling for various levels of memory hierarchy [2, 6, 13], and tiling for parallelism and locality [37, 5]. Multi-level tiling has become a key technique for high-performance computation. There has been work on generating efficient multi-level tiled code for polyhedral iteration spaces that handle tile sizes at compile time [23] and that handle tile sizes as symbolic parameters [26].

8. Conclusions

In this paper, we have developed approaches to address two main challenges in modern high-performance multi-level parallel architectures with explicitly managed scratchpad memories, namely, effective data management in scratchpad memories, and effective mapping of computation in regular programs on to multiple levels of parallel units on the architecture. We have developed a framework, to address the first problem, that automatically performs allocation of storage space in scratchpad memory to hold portions of data accessed in a computational block, determination of access functions of local memory array references, and generation of code for moving relevant portions of data resident in slow off-chip memory to fast scratchpad memory and vice versa. We have employed a multi-level tiling strategy, to address the second problem, that effectively handles the impact of on-chip memory availability on tile sizes at various levels. We have shown the effectiveness of our approach through experiments on GPUs which are representatives of high performance multi-level parallel architectures with explicitly managed scratchpad memories.

Acknowledgments This work is supported in part by the U.S. National Science Foundation through awards 0121676, 0121706, 0403342, 0508245, 0509442, 0509467 and 0541409.

References

[1] S. Anantharaman and S. Pande. Compiler optimizations for real time execution of loops on limited memory embedded systems. In *IEEE Real-Time Systems Symposium*, pages 154–164, 1998.

[2] Automatically Tuned Linear Algebra Software (ATLAS). http://math-atlas.sourceforge.net/.

[3] F. Balasa, P. Kjeldsberg, M. Palkovic, A. Vandecappelle, and F. Catthoor. Loop transformation methodologies for array-oriented memory management. In *17th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'06)*, pages 205–212, 2006.

[4] D. P. Bertsekas. Nonlinear Programming: 2nd Edition. Athena Scientific. ISBN 1-886529-00-0.

[5] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzaran, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP*, pages 48–57, 2006.

[6] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC. In *Proc. ACM International Conference on Supercomputing*, pages 340–347, 1997.

[7] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Affine transformations for communication minimal parallelization and locality optimization of arbitrarily nested loop sequences. Technical Report OSU-CISRC-5/07-TR43, Ohio State University, May 2007.

[8] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. Kjeldsberg, T. V. Achteren, and T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, 2002.

[9] P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze and transform scientific programs. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, pages 278–285, 1996.

[10] CLooG: The Chunky Loop Generator. http://www.cloog.org.

[11] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *IJPP*, 25(6):447–496, Dec. 1997.

[12] C. Eisenbeis, W. Jalby, D. Windheiser, and F. Bodin. A strategy for array management in local memory. In *Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*, pages 130–151, Irvine, Calif., 1990. Cambridge, Mass.: MIT Press.

[13] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[14] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988.

[15] P. Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 20(1):23–53, 1991.

[16] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *IJPP*, 21(5):313–348, 1992.

[17] P. Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *IJPP*, 21(6):389–420, 1992.

[18] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.

[19] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 328–343, London, UK, 1992. Springer-Verlag.

[20] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 229–254, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[21] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. FMI, University of Passau, 2004. Habilitation Thesis.

[22] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 49–52, 2006.

[23] M. Jimnez, J. M. Llabera, and A. Fernndez. A cost-effective implementation of multilevel tiling. *IEEE Trans. Parallel Distrib. Syst.*, 14(10):1006–1020, 2003.

[24] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. *IEEE Transactions on VLSI (TVLSI)*, 12(3):281–287, 2004.

[25] M. Kandemir, J. Ramanujam, M. Irwin, V. Narayanan, I. Kadayif, and A. Parikh. A compiler based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on Computer-Aided Design*, 23(2):243–260, 2004.

[26] D. Kim, L. Renganarayana, D. Rostron, S. Rajopadhye, and M. M. Strout. Multi-level tiling: M for the price of one. In *SC*, November 2007.

[27] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *ACM SIGPLAN PLDI 2007*, July 2007.

[28] A. Lim. *Improving Parallelism And Data Locality With Affine Partitioning*. PhD thesis, Stanford University, Aug. 2001.

[29] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL'97*, pages 201–214, 1997.

[30] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html.

[31] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandecappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Trans. Design Autom. Electr. Syst.*, 6(2):149–206, 2001.

[32] PolyLib - A library of polyhedral functions. http://icps.u-strasbg.fr/polylib/.

[33] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *CGO '07*, pages 144–156, 2007.

[34] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, Aug. 1992.

[35] W. Pugh. Counting solutions to presburger formulas: how and why. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 121–134, 1994.

[36] J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan. Reducing memory requirements of nested loops for embedded systems. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 359–364, 2001.

[37] L. Renganarayanan, M. Harthikote-Matha, R. Dewri, and S. V. Rajopadhye. Towards optimal multi-level tiling for stencil computations. In *IPDPS*, pages 1–10. IEEE, 2007.

[38] R. Schreiber and D. C. Cronquist. Near-Optimal Allocation of Local Memory Arrays. Technical Report HPL-2004-24, HP Laboratories Palo Alto, 2004.

[39] N. Vasilache, C. Bastoul, S. Girbal, and A. Cohen. Violated dependence analysis. In *ACM ICS*, June 2006.

[40] Y. Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 811–816, 1999.

Original Code:

```
A[200][200]; B[200][200];

for (i=10;i<=14;i++) {
  for (j=10;j<=14;j++) {
    A[i][j+1] = A[i+j][j+1]*3;
    for (k=11;k<=20;k++) {
      B[i][j+k] = A[i][k] + B[i+j][k];
    }
  }
}
```

Modified Code:

```
LA0[5][10]; LA1[9][5]; LB0[5][14]; LB1[9][10];

for (i=10;i<=14;i++) {
  for (j=10;j<=14;j++) {
    LA0[i-10][j+1-11] = LA1[i+j-20][j+1-11]*3;
    for (k=11;k<=20;k++) {
      LB0[i-10][j+k-21] = LA0[i-10][k-11] + LB1[i+j-20][k-11];
    }
  }
}
```

```
/* Array A*/
/* Local Memory Storage

 Local Array LA0:

 lb(i) = 10; ub(i) = 14
 lb(j) = 11; ub(j) = 20
 offset(i) = 10; offset (j) = 11;

 Local Array LA1:

 lb(i) = 20; ub(i) = 28
 lb(j) = 11; ub(j) = 15
 offset(i) = 20; offset (j) = 11;

*/
LA0[5][10]; LA1[9][5];

/* Data Move in code */
for (i=10;i<=14;i++) {
  for (j=11;j<=20;j++)
    LA0[i-10][j-11] = A[i][j] ;
}
for (i=20;i<=28;i++) {
  for (j=max(i-13,11);j<=min(15,i-9);j++)
    LA1[i-20][j-11] = A[i][j] ;
}

/* Data Move out code */
for (i=10;i<=14;i++) {
  for (j=11;j<=15;j++)
    A[i][j] = LA0[i-10][j-11];
}
```

```
/* Array B*/
/* Local Memory Storage

 Local Array LB0:

 lb(i) = 10; ub(i) = 14
 lb(j) = 21; ub(j) = 34
 offset(i) = 10; offset (j) = 21;

 Local Array LB1:

 lb(i) = 20; ub(i) = 28
 lb(j) = 11; ub(j) = 20
 offset(i) = 20; offset (j) = 11;

*/
LB0[5][14]; LB1[9][10];

/* Data Move in code */
for (i=20;i<=28;i++) {
  for (j=11;j<=20;j++)
    LB1[i-20][j-11] = B[i][j];
}

/* Data Move out code */
for (i=10;i<=14;i++) {
  for (j=21;j<=34;j++)
    B[i][j] = LB0[i-10][j-21];
}
```

Figure 1. Example of data allocation and movement

```
FORALL i = 1, Ni
  FORALL j = 1, Nj
    FOR k = 1, WS
      FOR l = 1, WS
        S1
      END FOR
    END FOR
  END FORALL
END FORALL
```

Figure 2. Example of Multi-level Tiling - Original Code

```
// Tiling to distribute at the outer level
FORALL iT = 1, Ni, Ti
  FORALL jT = 1, Nj, Tj
    // Tiling to satisfy local memory limit
    FOR i' = iT, min(iT+Ti−1,Ni), ti'
      FOR j' = jT, min(jT+Tj−1,Nj), tj'
        FOR k' = 1, WS, tk'
          FOR l'= 1, WS, tl'
            <Data move in Code>
            // Tiling to distribute at the inner level
            FORALL it = i', min(i'+ti'−1,Ni), ti
              FORALL jt = j', min(j'+tj'−1,Nj), tj
                FOR i = it, min(it+ti−1,Ni)
                  FOR j = jt, min(jt+tj−1,Nj)
                    FOR k = k', min(k'+tk'−1,WS)
                      FOR l = l', min(l'+tl'−1,WS)
                        S1
                      END FOR
                    END FOR
                  END FOR
                END FOR
              END FORALL
            END FORALL
            <Data move out Code>
          END FOR
        END FOR
      END FOR
    END FOR
  END FORALL
END FORALL
```

Figure 3. Example of Multi-level Tiling - Multi-level Tiled Code

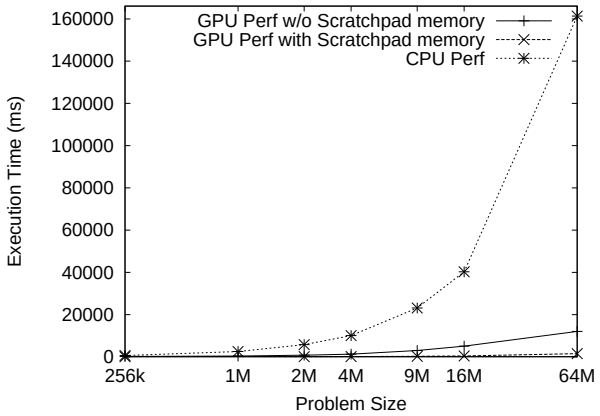

Figure 4. Execution time of Mpeg4 ME for various problem sizes

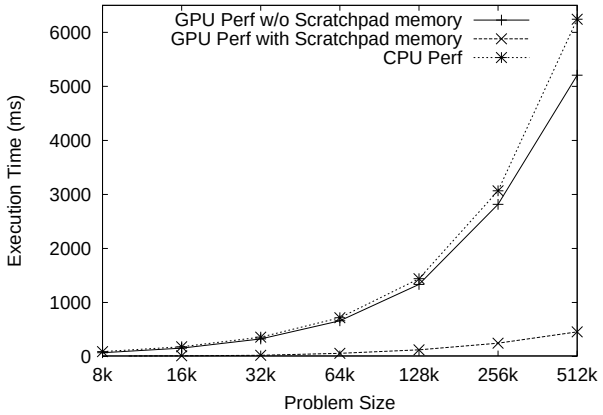

Figure 5. Execution time of 1-D Jacobi for various problem sizes

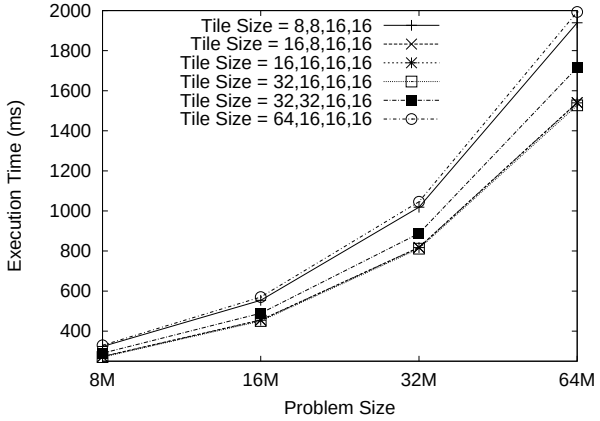

Figure 6. Execution time of Mpeg4 ME kernel for varying tile sizes

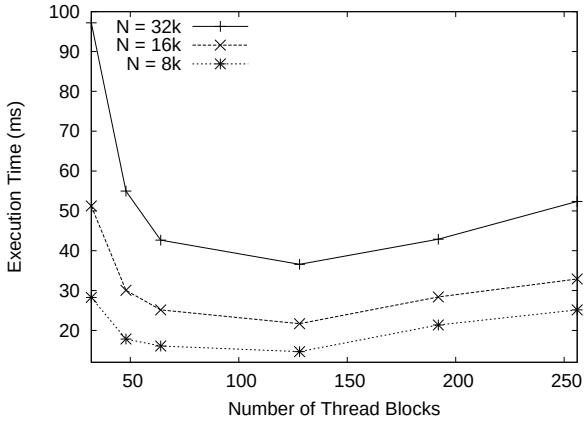

Figure 7. Execution time of 1-D Jacobi for smaller problem sizes for varying thread blocks

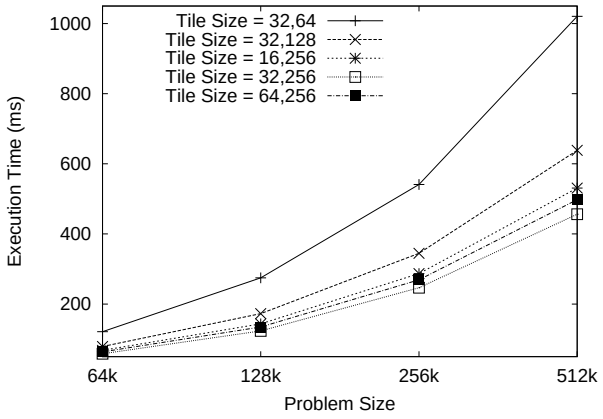

Figure 8. Execution time of 1-D Jacobi for larger problem sizes for varying tile sizes