In the Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-2011), San Jose, CA, June 4-8, 2011

Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors

Ashwin Prasad Jayvant Anantpur R. Govindarajan

Supercomputer Education and Research Centre,
Indian Institute of Science, Bangalore, India

ashwin@hpc.serc.iisc.ernet.in jayvant@hpc.serc.iisc.ernet.in govind@serc.iisc.ernet.in

Abstract

MATLAB is an array language, initially popular for rapid prototyping, but is now being increasingly used to develop production code for numerical and scientific applications. Typical MATLAB programs have abundant data parallelism. These programs also have control flow dominated scalar regions that have an impact on the program's execution time. Today's computer systems have tremendous computing power in the form of traditional CPU cores and throughput oriented accelerators such as graphics processing units(GPUs). Thus, an approach that maps the control flow dominated regions to the CPU and the data parallel regions to the GPU can significantly improve program performance.

In this paper, we present the design and implementation of MEGHA, a compiler that automatically compiles MATLAB programs to enable synergistic execution on heterogeneous processors. Our solution is fully automated and does not require programmer input for identifying data parallel regions. We propose a set of compiler optimizations tailored for MATLAB. Our compiler identifies data parallel regions of the program and composes them into kernels. The problem of combining statements into kernels is formulated as a constrained graph clustering problem. Heuristics are presented to map identified kernels to either the CPU or GPU so that kernel execution on the CPU and the GPU happens synergistically and the amount of data transfer needed is minimized. In order to ensure required data movement for dependencies across basic blocks, we propose a data flow analysis and edge splitting strategy. Thus our compiler automatically handles composition of kernels, mapping of kernels to CPU and GPU, scheduling and insertion of required data transfer. The proposed compiler was implemented and experimental evaluation using a set of MATLAB benchmarks shows that our approach achieves a geometric mean speedup of 19.8X for data parallel benchmarks over native execution of MATLAB.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Very high-level languages; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Algorithms, Languages, Performance

1. Introduction

Recent advances in the design of Graphics Processing Units (GPUs) [3][20][23] have significantly increased the compute capabilities of these processors. Today's desktops and servers, in addition to having multiple CPU cores, have graphics processing units with hundreds of SIMD cores. GPUs, due to their ability to accelerate data-parallel applications, are well suited for numerical and scientific computation. However, programming in these platforms remains a challenge as the programmer needs to manually coordinate tasks across the CPU cores and the GPU and manage data transfer. This requires him to have a detailed understanding of the machine architecture.

MATLAB [21], an array language, is popular for implementing numerical and scientific applications. MATLAB programs naturally express data-level parallelism and are therefore ideal candidates for acceleration using throughput-oriented processors like GPUs. Frequently, MATLAB programs are hand translated into compiled languages like C to improve performance. Manually translating MATLAB programs to use available accelerators requires the programmer to manage details such as data transfer and scheduling. This makes the translation a painstaking and error prone process. Therefore, a tool to automatically transform MATLAB programs so that accelerators are used to speedup the data-parallel parts of the program and the CPU is used to efficiently execute the rest of the program would be very useful.

In this paper, we present MEGHA[1], a compiler that automatically compiles MATLAB programs for execution on machines that contain both CPU and GPU cores. Prior work on compiling MATLAB has focussed on compiling it to languages such as FORTRAN [10] or C [17]. Just-in-time compilation techniques for execution of MATLAB programs on CPUs were implemented in MaJIC [2] and McVM [8]. However, these efforts on automatic translation are limited to CPUs and do not consider GPUs. Currently, Jacket [14] and GPUmat [12] provide the user with the ability to run MATLAB code on GPUs. These require users to specify arrays whose operations are to be performed on the GPU by declaring them to be of a special type. Further, the user needs to identify which operations are well suited for acceleration on the GPU, and is required to manually insert calls to move the data to and from the GPU memory.

Our compiler automatically identifies regions of MATLAB programs that can be executed efficiently on the GPU and tries to schedule computations on the CPU and GPU in parallel to further increase performance. There are several associated challenges. First, the CPU and GPU operate on separate address spaces requiring explicit memory transfer commands for transferring data from

[1] **M**ATLAB **E**xecution on **G**PU based **H**eterogeneous **A**rchitectures. *Megha* also means cloud.

and to the GPU memory. Second, sets of statements (kernels) in the input program that can be run together on the GPU need to be identified. This is required to minimize data transfers between host and device memory and also to reduce kernel call overheads. Third, program tasks [2] need to be mapped to either GPU cores or CPU cores and scheduled so that the program execution time is minimized. Required data transfers also need to be taken into account while mapping and scheduling tasks. Lastly, efficient code needs to be generated for each task depending on which processor it is assigned to.

The following are the main contributions of this work.

- We present a compiler framework, MEGHA, that translates MATLAB programs for synergistic parallel execution on machines with GPU and CPU cores.
- We formulate the identification of data parallel regions of a MATLAB program as a constrained graph clustering problem and propose an efficient heuristic algorithm to solve the problem.
- We propose an efficient heuristic algorithm to map kernels to either the CPU or GPU and schedule them so that memory transfer requirements are minimized. Techniques for generating efficient code for kernels mapped to either the CPU or GPU are also described.
- Experimental evaluation on a set of MATLAB benchmarks shows a geometric mean speedup of 19.8X for data parallel benchmarks over native MATLAB execution. Compared to manually tuned GPUmat versions of the benchmarks, our approach yields speedups up to 10X.

To the best of our knowledge, MEGHA is the first effort to *automatically* compile MATLAB programs for parallel execution across CPU and GPU cores. The rest of this paper is organized as follows: Section 2 briefly reviews the necessary background. Section 3 provides an overview of the compiler and briefly describes various stages in the frontend. Section 4 describes the details of our proposed compilation methodology. In Section 5, we present our experimental methodology and report performance results. Section 6 discusses related work and Section 7 concludes.

2. Background

2.1 NVIDIA GPUs and CUDA

This section describes the architecture of NVIDIA GPUs in general. While models might differ in the amount of resources available, the basic architecture stays roughly the same [22][20]. The architecture of NVIDIA GPUs is hierarchical. They consist of a set streaming multiprocessors (SMs) each of which consists of several scalar units (SUs). For example, the 8800 series GPUs have 16 SMs each having 8 SUs. The basic schedulable entity for GPUs is a *warp* which currently consists of 32 contiguous threads. SMs have support for the time interleaved execution of several warps. They periodically switch between warps to hide latency. Within an SM, all SUs execute instructions in lock-step. Additionally, any divergence in the execution path of threads within a warp results in a performance penalty. Each SM has a partitioned register file and a *shared memory* which is accessible by all SUs in the SM. The shared memory is similar to a software managed cache or a scratch pad memory. The 8800 series GPUs have register banks containing 8192 32-bit registers and 16KB of shared memory.

All SMs are connected to the *device memory* by a very wide bus. This bus is 256-512 bits wide for 8800 series GPUs depending on the model. The memory bus is capable of providing very high bandwidth, provided all threads in a warp access consecutive memory locations. Such accesses are said to be *coalesced*. Essentially the address accessed by thread n of a warp must be of the form $BaseAddress + n$.

Programs to be run on NVIDIA GPUs are typically written using the CUDA(Compute Unified Device Architecture) programming language which is an extension of C++ [22]. The CUDA programming model provides a higher level view of the GPU architecture to application programmers. From the programmer's point of view, threads are divided into *thread blocks*, and several thread blocks form a *grid*. A GPU *kernel* call, which is executed on the GPU, in general consists of multiple thread blocks organized as a grid. Each thread block is executed on a single SM and can contain up to 512 threads. However, a single SM can execute multiple thread blocks simultaneously depending on the register and shared memory requirement per block. Kernel calls are asynchronous in the sense that control is returned to the calling CPU thread before the kernel execution on the GPU completes. Data transfers between GPU and CPU memory must be initiated by the CPU thread.

2.2 Supported MATLAB Subset

MATLAB is a dynamically typed array based programming language which is very popular for developing scientific and numerical applications. As MATLAB has grown into a diverse and vast language over the years, the compiler described in this paper supports only a representative subset of MATLAB. A brief description of the subset of MATLAB supported and a set of other assumptions made by our compiler implementation are presented below.

1. MATLAB supports variables of several primitive types like `logical`, `int`, `real`, `complex` and `string`. It is also possible to construct arrays of these types with any number of dimensions. Currently, our compiler does not support variables of type `complex` and `string`. Arrays are also restricted to a maximum of three dimensions.
2. MATLAB supports indexing with multi-dimensional arrays. However, our implementation currently only supports indexing with single dimensional arrays.
3. In MATLAB, it is possible to change the size of arrays by assigning to elements past their end. We currently do not support indexing past the end of arrays. Further, in this paper, we refer to assignments to arrays through indexed expressions (Eg: `a(i)`) as indexed assignments or *partial assignments*.
4. We assume that the MATLAB program to be compiled is a single script without any calls to user-defined functions. Support for user defined functions can be added by extending the frontend of the compiler. Also, anonymous functions and function handles are not currently supported.
5. In general, types and shapes (array sizes) of MATLAB variables are not known until run-time. The compiler currently relies on a simple data flow analysis to extract sizes and types of program variables. It also relies on programmer input when types cannot be determined. We intend to extend our type system to support symbolic type inferencing in future [16]. Ultimately, we envision that the techniques described in this paper will be used in both compile time and run-time systems.

3. Compiler Overview

A high level overview of the proposed compiler is shown in Figure 1. The input to the compiler is a MATLAB script and the output is a combination of C++ and CUDA code. This section briefly describes the frontend of the compiler. A detailed explanation of the stages in the backend is given in Section 4.

[2] Statements that can only run on the CPU and kernels (which can be run on CPU or GPU) are collectively called tasks.

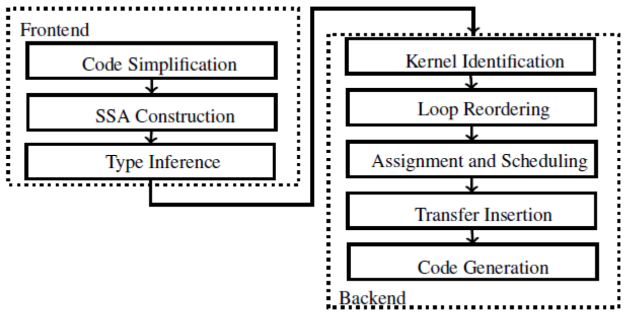


Figure 1: Compiler Overview

3.1 Code Simplification

The compiler first simplifies the input source code by breaking complex expressions into simpler ones. The output of this pass is a single operator intermediate form; i.e., each statement in the resulting IR has at most one operator on the right hand side of assignments. Expressions are broken into single operator form by introducing temporaries. For example,

```
x = (a + b) - c;
```

is converted to

```
tempVar1 = a + b;
x = tempVar1 - c;
```

Array indexing is also treated as an operator. First each indexing expression is simplified if needed. Then a simplified array reference is emitted. For example,

```
x = a(2*i);
```

is converted to

```
tempVar1 = 2*i;
x = a(tempVar1);
```

However, it is not always possible to completely simplify indexing expressions. This is the case when indexing expressions contain *magic ends* (e.g., `a(2:end)` refers to all elements of `a` starting from the second element) and *magic colons* (e.g., `a(:)` refers to all elements of `a`). These are handled using a set of semantics preserving transformations described below.

3.2 Semantics Preserving Transformations

Transformation of the MATLAB code is necessary to simplify subsequent analysis, enable conversion to single assignment form and ultimately enable CUDA code generation. Hence, our compiler makes the following changes to the simplified IR before converting it to SSA form.

MATLAB supports "context-sensitive" indexing of arrays. Users can specify all elements along a dimension by simply using a ":" in the index corresponding to that dimension. For example, to set all elements of a vector "a" to 42, one could write:

```
a( : ) = 42
```

One can also use the keyword `end` inside an array index expression in MATLAB to refer to the last element of the array along that dimension. For example, if "a" is a vector,

```
a(4:end) = 42;
```

assigns 42 to all elements of "a" starting from element 4. The compiler removes these context-sensitive operators by replacing them by the indices they imply : e.g., `a(:)` is replaced by `a(1:length(a))` and `a(2:end)` is replaced by `a(2:length(a))`.

Some transformations are also needed on for-loops in user code to preserve MATLAB semantics. In MATLAB, assignments to the loop index variable or to the loop bounds inside the loop body do not have any effect on the iteration count of the loop. For example, the loop in the following program runs for ten iterations.

```
n = 10;
for i=1:n
  % ....
  i = i + 10;
endfor
```

Our compiler preserves these semantics by renaming the loop index (and bounds). A statement to copy the new loop index into the old loop index is inserted at the head of the loop. Therefore, the above code would be transformed to the following equivalent code.

```
n = 10;
for tempIndex=1:n
  i = tempIndex;
  % ....
  i = i + 10;
endfor
```

3.3 Static Single Assignment Construction

Since static single assignment form (SSA) is a well established IR for performing optimizations, we choose it as our intermediate form. The SSA form is constructed using the method described by Cytron et al [9]. However, since the variables in the program are possibly arrays, it is important to optimize the insertion of ϕ nodes. One optimization we make is that we insert ϕ nodes for a variable at a point only if it is live at that point. Further, assignments to array variables through indexed expressions are not renamed; in other words, values are assigned in-place. However, all total assignments are renamed.

3.4 Type and Shape Inference

Since MATLAB is a dynamically typed language, type and shape information needs to be inferred before a MATLAB script can be compiled to a statically typed language like C. Our definition of the type of a variable is based on the definition used by De Rose et al [10]. The type of a variable is a tuple consisting of the following elements

1. *Intrinsic Type* which can be `boolean`, `integer` or `real`,
2. *Shape* which can be `scalar`, `vector`, `matrix` or `3D-array`, and
3. *Size* which indicates the size of each dimension of the variable.

Our compiler currently tries to infer base types of variables and shapes of arrays based on program constants and the semantics of built-in MATLAB functions and operators by performing a forward data flow analysis that is similar to what was used in FALCON [10] and MaJIC [2]. It is also possible for the user to specify types of variables in the input MATLAB script via annotations. The compiler does not currently implement symbolic type inference and symbolic type equality checking [16]. We leave this for future work.

4. Backend

The backend of the compiler performs kernel identification, mapping, scheduling and data transfer insertion. Kernel identification identifies sets of statements, called *kernels*[3], which can be treated as a single entity by the mapping and scheduling phase. It also transforms kernels into loop nests from which the code generator can generate efficient lower level code. The mapping and scheduling phase assigns a processor to each identified kernel and also decides when it should be executed. Kernel identification and mapping are performed per basic block. Global transfer insertion then performs a global data flow analysis and inserts the required inter basic block transfers.

4.1 Kernel Identification

The purpose of the kernel identification process is to identify the entities on which mapping and scheduling decisions can be made. We call these entities *kernels* or tasks[4]. They are sets of IR statements that, when combined, can be efficiently run on the GPU. The kernel identification process consists of the following steps.

1. Identify IR statements that are data parallel as "GPU friendly" statements.
2. *Kernel Composition* identifies sets of statements that can be grouped together in a common kernel.
3. *Scalarization* and *Index Array elimination* eliminate arrays that are not needed because of the way statements were grouped together.
4. *Surrounding Loop Insertion* transforms kernels into loop nests consisting of parallel loops.

Our compiler currently tags statements that perform element-wise operations or other data parallel operations like matrix multiplication as "GPU friendly" statements. It should be noted that this step just identifies statements that *could* run on the GPU and does not make any decisions on what should actually be run on the GPU.

```
1: A = (B + C) + (A + C);
2: C = A * C;
```

(a) MATLAB Statements

```
1: tempVar0 = B + C;
2: tempVar1 = A + C;
3: A_1 = tempVar0 + tempVar1;
4: tempVar2 = A_1 * C;
5: C_1 = tempVar2;
```

(b) IR Representation

Figure 2: A simple MATLAB program and its IR representation. A, B and C are 100x100 matrices.

Once GPU friendly statements have been identified, the kernel composition step decides the granularity at which mapping decisions are to be made. It does this by grouping IR statements together. It is inefficient to make these decisions at the level of IR statements as there may be too many of them. Kernel composition reduces the number of entities that need to be mapped and could also lead to a significant decrease in the memory requirement of the compiled program. To understand why, consider the example in Figure 2(a). Code simplification generates a considerable number of temporaries that are used only once after they are defined as can be seen in Figure 2(b). If mapping decisions are made per IR statement, the definition and use of these temporaries could be mapped to different processors necessitating the storage of these temporaries, which may be arrays, in memory. In the example in Figure 2(b), three temporaries, each of which is a matrix have been generated by the simplification process. In the worst case, when all these temporaries need to be stored in memory, the memory requirement of the program increases significantly. However, if statements 1, 2 and 3 are grouped into the same kernel, it is possible to replace the arrays tempVar0 and tempVar1 with scalars. When this kernel is rewritten as a loop nest, these variables are only live within one iteration of the loop nest. Hence they can be replaced by scalars as in the following code.

```
for i=1:100
  for j=1:100
    tempVar0_s = B(i, j) + C(i, j);
    tempVar1_s = A(i, j) + C(i, j);
    A_1(i, j) = tempVar0_s + tempVar1_s;
  endfor
endfor
```

The above problem is even more exaggerated in real benchmarks. To give a specific example, in the benchmark fdtd [15], we observed that worst case memory requirement increase can be as high as 30X. Combining statements into kernels is also important because the mapping and scheduling phase would then be able to solve a scheduling problem with a fewer number of nodes.

We refer to this process of reducing arrays to scalar variables as *Scalarization*. However, the newly created scalars are likely to be saved in registers thereby increasing the register utilization.

Two statements can be combined into a kernel only if the following conditions hold for the statements to be combined.

1. Both statements perform element-wise operations.
2. Both statements have identical iteration spaces. (By iteration space of a statement, we mean the iteration space of the parallel loops surrounding the statement.)
3. Combining the statements will not result in cyclic dependencies among kernels.
4. The loops surrounding the kernel created by combining the two statements should be parallel[5].

In the example in Figure 2(b), it is legal to group the first three statements, in the simplified IR, into the same kernel. However, it is illegal to put statement 4 into the same group as matrix multiplication is not an element-wise operation and requires elements at other points in the iteration space to have been computed. Thus, combining the addition and the multiplication statements introduces cross iteration dependencies and therefore the surrounding loops of the resulting group are no longer parallel. Even though condition 1 is subsumed by condition 4, it is specified separately as it is easier to check for violations of condition 1.

Condition 4 is similar to the condition for legal loop fusion [18] but is stricter. For loop fusion to be legal, it is only required that directions of dependencies between statements in the loops being fused are not reversed because of the fusion. However, for kernel composition to be legal, we require that the surrounding loops still be parallel loops after the statements are combined as our objective is to map identified kernels to the GPU.

Combining statements into kernels has many benefits. Compiler generated temporary arrays and user arrays can be converted to scalars. Locality between the combined statements can be exploited. Lastly, combining statements into kernels also results in a reduction in the CUDA kernel invocation overhead at run-time. However, forming larger kernels can also increase register usage and the memory footprints of kernels. These factors are extremely

[3] We use "kernel" to refer to either the smallest unit of data-parallel work in a set of IR statements or a set of GPU friendly IR statements. Which is meant will be clear from context.

[4] The term kernel or task is used depending on whether they are mapped to the GPU or CPU respectively. We use both terms interchangeably when the mapping is not known.

[5] Parallel loops are loops whose iterations can all be run in parallel.

important if the kernel is to be run on the GPU. Larger kernels may also require more variables to be transferred to and from the GPU. This may lead to decreased opportunities for overlapping computation and data transfer. Thus there are trade offs in kernel composition.

We model the kernel composition problem as a constrained clustering problem modelling the costs and benefits described above. It is loosely related to the parameterized model for loop fusion [24]. However, compared to the model for loop fusion, our formulation additionally models memory utilization of kernels. Register utilization is also more accurately modelled. The details of the construction and a heuristic algorithm to compute a solution are described in the following sub-sections.

4.1.1 Graph Formulation

The problem of composing statements into kernels is that of clustering the nodes of an augmented data flow graph $G = (V, E)$ into clusters. A cluster is a set of nodes of the graph G. The graph G is defined as follows.

1. Each node $n \in V$ represents an IR statement in the basic block under consideration. The statement represented by $n \in V$ is denoted by $stm(n)$.
2. The set of edges, E has two types of edges.
    - *Dependence edges* are directed edges between nodes whose statements are data dependent. An edge (n_1, n_2) implies that n_2 must run after n_1. All types of dependences (true, anti and output) are represented uniformly.
    - A *Fusion preventing edge* connects a pair of nodes that cannot be merged. For example, such an edge would exist between the vertices for statements 3 and 4 in Figure 2(b). There are also fusion preventing edges between nodes representing GPU friendly statements and others representing non GPU friendly statements.

The kernel composition problem is that of clustering the set of nodes V into a set of clusters $\{C_1, C_2, ...C_n\}$. All the C_i's are disjoint and their union is V. An edge $e = (n_1, n_2)$ is said to be in a cluster C_i if n_1 and n_2 are in C_i.

To accurately model register utilization, we consider the original register usage of each statement in the basic block and the increase in register usage due to scalarization of arrays. First, the register utilization of each statement $stm(n)$, for each $n \in V$, denoted by reg_n, is estimated as the number of scalars used in $stm(n)$. Secondly, we need to differentiate between variables that can be scalarized (scalarizable variables) and those that cannot be scalarized by combining two statements. To model the increase in register utilization when two statements are combined, we define the weight reg_e on each edge $e = (n_1, n_2) \in E$. The value of this quantity is the number of variables that can be scalarized by combining $stm(n_1)$ and $stm(n_2)$. For example, it is 1 for the edge between the nodes for statements 1 and 3 in the above example since `tempVar0` can be eliminated by combining these statements. Now, when a cluster is formed, the register utilization of the kernel represented by the cluster is the sum of the reg values of all nodes and edges in the cluster. Therefore, to limit the register utilization of each cluster, we introduce the following constraint for each cluster C_i.

$$\sum_{e \in C_i} reg_e + \sum_{n \in C_i} reg_n \leq R_{max} \tag{1}$$

To model memory utilization, the distinction between scalarizable variables and non-scalarizable variables is needed again. For scalarizable variables, such as `tempVar0`, the memory utilization becomes zero when its defining node and use node are combined (statements 1 and 3 in this case). However, for non-scalarizable variables, the memory usage is the sum of the sizes of all such variables used by statements in the cluster.

To model the memory utilization due to scalarizable variables, we use weights on both edges and nodes. For a node n, we define mem_n as the sum of sizes of all scalarizable variables in $stm(n)$. Therefore mem_n, for the node representing statement 1 is $size(tempVar0)$. For an edge e between n_1 and n_2, if there is a variable $scal\_var$ that can be scalarized by combining $stm(n_1)$ and $stm(n_2)$, we define $mem_e = -2.size(scal\_var)$. Therefore, the memory utilization of a scalarizable variable reduces to zero if both its definition and use are in the same cluster as the weights on the nodes are cancelled out by the weight of the edge between them. The total memory usage of the kernel represented by the cluster C_i, due to scalarizable variables is therefore given by:

$$\sum_{n \in C_i} mem_n + \sum_{e \in C_i} mem_e = M_{scal,i} \tag{2}$$

For non-scalarizable variables, we use a bit vector to model the memory utilization. For each node $n \in V$, and each non-scalarizable variable var, we define $used_n(var)$ to be 1 if n uses var. For example, the node m, representing statement 1 in the above example, has $used_m(B) = 1$ and $used_m(C) = 1$. A variable var is used in a cluster if any of the nodes in the cluster has $used(var)$ as 1. The memory utilization of a cluster due to non-scalarizable variables is the sum of sizes of all such variables used in the cluster. The memory utilization due to non-scalarizable variables is given by:

$$\sum_{var \in Vars} (\bigvee_{n \in C_i} used_n(var)).size(var) = M_{nonscal,i} \tag{3}$$

where $Vars$ is the set of all non-scalarizable variables in the program.

To limit the total memory utilization of each kernel, we introduce the following constraint.

$$M_{scal,i} + M_{nonscal,i} \leq M_{max} \tag{4}$$

Additionally, for all fusion preventing edges $e \in E$, e must not be in any cluster C_i. Also, when nodes are grouped together as kernels, the graph must remain acyclic.

For each edge $e = (n_1, n_2) \in E$, $benefit_e$, quantifies the benefit of combining the statements $stm(n_1)$ and $stm(n_2)$ into the same kernel. The benefit of collapsing an edge is defined as

$$benefit_e = \alpha Mem(n_1, n_2) + \beta Loc(n_1, n_2) + \gamma$$

where, $Mem(n_1, n_2)$ is the sum of sizes of array variables that can be eliminated by combining $stm(n_1)$ and $stm(n_2)$, $Loc(n_1, n_2)$ quantifies the locality between the two statements. It is currently estimated as the number of pairs of overlapping array references in the two statements. If two statements being considered for composition have a common array reference, then this can be read into shared memory and reused. The term $Loc(n_1, n_2)$ is meant to model this. γ represents the reduction in the kernel call overhead. α and β are parameters.

The objective of kernel composition is to maximize the total $benefit$:

$$benefit = \sum_{C_i} \sum_{e \in C_i} benefit_e$$

subject to the above constraints. The above problem can be shown to be NP-hard by a reduction from the loop fusion problem defined by Singhai and McKinley [24]. We therefore use a heuristic algorithm to solve the kernel composition problem.

4.1.2 Kernel Composition Heuristic

Our heuristic kernel clustering method uses a k-level look ahead to merge nodes of the graph G into clusters. It computes the benefits and memory requirement for collapsing all possible legal sets of k edges starting at node n by exhaustively searching through all possibilities. (By collapsing an edge, we mean combining its source and destination nodes). It then collapses the set of k edges that results in the maximum benefit. For small values of $k(k = 3$ or $k = 4)$, this achieves reasonably good results without a significant increase in compile time. The details of the clustering algorithm are listed in Algorithm 1. The function $legalKDepthMerges$ returns multiple sets of (up to) k edges that can be *legally* collapsed. This function makes sure that clusters that have fusion preventing edges are not formed and that no cycles are formed in the graph. It also ensures that the constraints listed earlier are not violated. The function $maxBenefit$ returns the set of edges with the maximum benefit and $collapseEdges$ collapses all edges in a set of edges and adds all combined nodes to C_i.

Algorithm 1 Kernel Composition Heuristic

```
procedure KernelComposition($G = (V, E)$)
nodes ← V
clusters ← $\phi$
while $nodes \neq \phi$ do
    $n \leftarrow$ node in $nodes$ that uses minimum memory
    $C_i \leftarrow n$
    $nodes \leftarrow nodes - \{n\}$
    while $true$ do
        $T \leftarrow legalKDepthMerges(C_i, k)$
        if $T \neq \phi$ then
            $bestSet \leftarrow maxBenefit(T)$
            $collapseEdges(C_i, bestSet)$
            for all $e = (n_1, n_2) \in bestSet$ do
                $nodes \leftarrow nodes - \{n_1\}$
                $nodes \leftarrow nodes - \{n_2\}$
            end for
        else
            break
        end if
    end while
    $clusters \leftarrow clusters \cup C_i$
end while
end procedure
```

4.1.3 Identifying In and Out Variables

Once kernels have been identified, we need to compute the set of variables that are required for the execution of the kernel and the set of variables that are modified by the kernel and are needed by future computations. To do this we first perform a live-variable analysis [1] on the SSA IR. A variable is an "in-variable" of a given kernel if it is live on entry to the kernel and is accessed (either read or partially written) within the kernel. We define a variable as an "out-variable" if the variable is modified inside the kernel and is live at the exit of the kernel. Data transfer from CPU memory or GPU memory may be required for in-variables and out-variables depending on the processors kernels are ultimately mapped to.

4.1.4 Array Elimination

As discussed in Section 4.1, arrays that become unnecessary after kernel composition need to be eliminated. Replacing these arrays with scalar variables not only reduces memory requirements but also eliminates memory accesses that are expensive, especially if they are serviced out of GPU device memory. To remove array variables, we use the two transformations, scalarization and index array elimination, described below.

Scalarization

After in-variables and out-variables of each kernel have been identified it is possible to replace an array that is neither an in-variable nor an out-variable by a scalar as shown in the earlier example. This is possible because it is guaranteed that no subsequent statement depends on writes to the array within this kernel and that this kernel does not depend on the prior value of this array. We refer to this process as *scalarization*.

Index Array Elimination

In MATLAB programs, arrays (especially those defined using the colon operator) are frequently defined and used either for indexing or in element-wise computations. It is possible to replace some references to these arrays with scalars that are computed inside the kernel in which the original array was used. Any array representable as a function $f: \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ that maps the index of an array element to the element value and used to index one or more dimensions of another array is called an *index array*. A total reference to such an array in a kernel can be replaced by a scalar that takes the value $f(i)$ in the i^{th} iteration of the surrounding loop. Consider the following example, in which a subset of the 3D array `Ey` is copied into `tempVar1`.

```
tempVar0 = Nz + 1;
tempVar1 = 2:tempVar0;
tempVar2 = Ey(:, :, tempVar1);
```

Here, `tempVar1` is an index array with the function f given by $f(i) = 2 + (i - 1)$. It can be eliminated by replacing it with a scalar while adding surrounding loops as shown below (details in Section 4.1.5).

```
tempVar0 = Nz + 1;
for i = 1:Sx
  for j = 1:Sy
    for k = 1:Sz
      indexVar1 = 2 + (k-1);
      tempVar2(i, j, k) = Ey(i, j, indexVar1);
    end
  end
end
```

`Sx`,`Sy` and `Sz` represent the size of the iteration space of the original assignment in the first, second and third dimension respectively. We refer to this transformation as *index array elimination*.

Currently, our compiler identifies arrays defined using the colon operator as index arrays. However, arrays defined using functions like `linspace` [6] and in user loops could also be identified as index arrays. We intend to implement these extensions in the future.

4.1.5 Surrounding Loop Insertion

Before code can be generated, identified kernels need to be transformed into a form more suitable for code generation. The compiler achieves this by adding loops around kernels and introducing the necessary indexing.

For kernels that contain element-wise operations, the required number of surrounding loops are inserted and array references are changed as described next. Consider the example in Section 4.1.4. Introducing indexing is simple for originally unindexed references like `tempVar2`. The indices are just the loop variables of the surrounding loops. However, the case for indexed references is slightly more complicated. The n^{th} *non-scalar* index in the indexed reference is transformed according to the following rules.

[6] `linspace` is a MATLAB function that generates linearly spaced vectors. `linspace(x, y, n)` generates `n` linearly spaced numbers from `x` to `y`. For example, `linspace(1, 9, 3)` returns the array `[1 5 9]`.

1. If it is a magic colon, replace it by i_n that is the loop variable of the n^{th} surrounding loop. For example, the first and second indexing expressions in the reference to `Ey` in the above example are changed to `i` and `j`.
2. If the indexer is an index array, replace the array by a scalar variable initialized with the value $f(i_n)$, where f is the function described in Section 4.1.4. This rule is used to change the third indexing expression in the reference to `Ey` from `tempVar2` to `indexVar1`.
3. Otherwise, if the n^{th} non-scalar indexer is x, then replace it by $x(i_n)$.

Once surrounding loops are inserted, the body of the loop nest represents exactly the CUDA code that would need to be generated if the kernel under consideration gets mapped to the GPU.

Kernels that contain more complex operators such as matrix multiplication are treated as special cases when they are transformed into loop nests. The compiler also keeps the original high-level IR as this is useful during code generation (for example to map a matrix multiplication kernel to a BLAS library call).

4.2 Parallel Loop Reordering

After the steps described in the previous section have been performed, the IR contains a set of kernels and their surrounding parallel loops. The in-variables and out-variables of each kernel are also known. When these kernels are ultimately executed, the way in which the iteration space of the surrounding loops is traversed can have a significant impact on performance irrespective of whether the kernel is mapped to the CPU or the GPU. Further, the iteration space may need to be traversed differently depending on whether the kernel is mapped to the CPU or the GPU. Since the surrounding loops are parallel, the iteration space of these loops can be traversed in any order. The loop reordering transform decides how the iteration space must be traversed to maximize performance.

For a kernel that is executed on the CPU, locality in the inner most loop is important from a cache performance viewpoint. For example, consider the code listed in Section 4.1.4. There are three surrounding loops with indices `i`, `j` and `k`. Therefore, the two indexed references to `tempVar2` and `Ey` will access consecutive memory locations for consecutive values[7]of `j`. This implies that having the `j` loop as the innermost loop will maximize locality.

GPU memory optimizations are however very different from memory optimizations implemented by the CPU. When all threads in a warp access a contiguous region of memory, the GPU is able to coalesce [22] these multiple memory requests into a single memory request thereby saving memory bandwidth and reducing the overall latency of the memory access. If a kernel is assigned to the GPU, the code generator maps the index of the outer most loop to the `x` dimension of the thread block. The GPU groups neighboring threads in the `x` dimension into the same warp. Therefore, in the GPU case, the loop with the maximum locality needs to be the outermost loop (which in the above example is the `j` loop).

At this stage, since it is not known which processor, CPU or GPU, a kernel is mapped to, the compiler computes loop orders for both of them. Also, both versions are needed during the profiling stage. Currently, a simple heuristic is used to compute the loop ordering. The number of consecutive memory references in successive iterations of each loop (with all other loop indices staying constant) is computed. The loops are then sorted based on this number. The loop that causes accesses to consecutive memory locations for the most references in the kernel is treated as the loop with the maximum locality.

For the CPU loop order, the loops are sorted in increasing order of locality. (The first loop in the sorted sequence is the outermost loop and the last loop is the inner most loop.) For the example above, the `j` loop has the most locality since both indexed references access successive memory locations for successive values of `j`. The other loops have no locality. Therefore, the `j` loop becomes the innermost loop when the above kernel is executed on the CPU.

For the GPU, the loop with the maximum locality becomes the outer-most loop since this enables coalescing for the greatest number of accesses. If a particular kernel is mapped to the GPU, final code generation maps the iteration number in the outermost loop in the GPU order to the thread ID in the X direction. It generates thread blocks of size (32, 1, 1). Since CUDA supports only 2 dimensional grids, the iteration space of the next two loops (if they exist) is flattened into the block ID in the y direction. In the above example, `j` becomes the outer most loop and the `i` and `k` loop get flattened into a single loop as shown below.

```
tempVar0 = Nz + 1;
for tidx = 1:Sy
  for tidy = 1:Sx*Sz
    j = tidx;
    i = floor(tidy/Sz);
    k = tidy % Sz;
    indexVar1 = 2 + (k-1);
    tempVar2(i, j, k) = Ey(i, j, indexVar1);
  end
end
```

It is possible that changing the layout of certain arrays depending on the surrounding loops may improve performance by either improving locality on CPUs or increasing the number of coalesced accesses on the GPU. We leave this problem for future work.

4.3 Mapping and Scheduling

Once kernels have been identified by the preceding steps of the compiler, each basic block can be viewed as a directed acyclic graph (DAG) whose nodes represent kernels. Edges represent data dependencies. It is now required to assign a processor and a start time to each node in the DAG such that the total execution time of the basic block under consideration is minimized. This problem is similar to the traditional resource constrained task/instruction scheduling problem except that some tasks can be scheduled either on the CPU or GPU. Therefore, when dependent tasks are mapped to different types of processors, required data transfers also need to be inserted. This makes the above problem harder than the scheduling problem. It is therefore NP-hard. Thus, we propose a heuristic approach for the integrated mapping and scheduling problem.

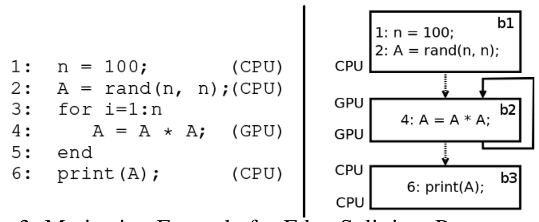
The compiler uses a variant of list scheduling to assign nodes to processors. The heuristic is based on the observation that a node whose performance is very good on one processor and very bad on the other needs to be given maximum chance to be assigned to the processor on which it performs best. We define $skew(n)$ where n is a node in the DAG as

$$skew(n) = max\left(\frac{T_{GPU}(n)}{T_{CPU}(n)}, \frac{T_{CPU}(n)}{T_{GPU}(n)}\right)$$

where T_{GPU} and T_{CPU} are functions that give the execution time of n on the GPU and CPU respectively. To obtain estimates for the execution time for each node, we currently use profiling information.

The algorithm maintains resource vectors for the GPU, CPU and the bus. In every iteration, it gets the node from the ready queue with the highest skew and tries to schedule it so that the finish time is minimized. While scheduling a node, transfers required to

[7] Currently, in our implementation, three dimensional arrays are stored as stacked row major matrices in the spirit of MATLAB. Therefore, `tempVar2(i, j, k)` and `tempVar2(i, j+1, k)` are in successive memory locations.



```
1:  n = 100;        (CPU)
2:  A = rand(n, n);(CPU)
3:  for i=1:n
4:      A = A * A;  (GPU)
5:  end
6:  print(A);       (CPU)
```

Figure 3: Motivating Example for Edge Splitting. Processor name in brackets indicates mapping. The control flow graph shows split edges as dotted lines and the location of A at each program point.

start the node need to be considered. The algorithm computes the earliest time at which the node can start on the GPU and the CPU considering the data transfers that need to be performed so that all required data is available. The finish time of the node is computed for both the CPU and the GPU and the node is then scheduled on the processor on which it has the lower finish time. The details are given in Algorithm 2.

In the algorithm, $readyQueue$ is a priority queue that is prioritized based on $skew$. The function $computeMinimumStartTime$ computes the minimum time at which a node can start on a given processor considering the data transfers that are needed due to prior processor assignments and the resource availability. The functions $scheduleOnGPU$ and $scheduleOnCPU$ schedule a node onto the GPU and CPU respectively. They also insert and schedule the required data transfers and update the resource vectors.

Algorithm 2 Processor Assignment

```
1:  procedure Assignment (DataFlowGraph $G = (V, E)$)
2:  $readyQueue \leftarrow \phi$
3:  $\forall v \in V$ with no in edges : $readyQueue.insert(v)$
4:  while $readyQueue \neq \phi$ do
5:      $v \leftarrow readyQueue.remove()$
6:      $start_{GPU} \leftarrow computeMinimumStartTime(v, GPU)$
7:      $start_{CPU} \leftarrow computeMinimumStartTime(v, CPU)$
8:      $finish_{GPU} \leftarrow start_{GPU} + T_{GPU}(v)$
9:      $finish_{CPU} \leftarrow start_{CPU} + T_{CPU}(v)$
10:     if $finish_{GPU} \leq finish_{CPU}$ then
11:         $scheduleOnGPU(v, start_{GPU})$
12:     else
13:         $scheduleOnCPU(v, start_{CPU})$
14:     end if
15:     $V = V - v$
16:     for all $v_1 \in V$ with all preds scheduled do
17:         $readyQueue.insert(v_1)$
18:     end for
19: end while
20: end procedure
```

4.4 Global Data Transfer Insertion

All the previously described optimizations are performed per basic block. The mapping heuristic assumes that data which is read before it is written in a basic block is available on both the GPU and CPU. However, where the data actually is when the basic block begins executing depends on the program path that was taken to get to the basic block.

Consider the code segment in Figure 3. When execution reaches statement 4, matrix A may either be in the CPU memory or GPU memory depending on the path the program took to get to statement 4. If the path taken was 1→2→4, then A is in CPU memory since statement 2 runs on the CPU. However, if the path taken to reach statement 4 was 1→2→4→4, then A is in GPU memory since statement 4 runs on the GPU. Additionally, statement 4 assumes that A is available on the GPU.

To remove this dependence of a variable's location on the program path taken, we use a two step process. We first use a data flow analysis to compute the locations of all program variables at the entry and exit of each basic block in the program. Then, we split control flow edges whose source and destination have variables in different locations.

4.4.1 Variable Location Analysis

The aim of the data flow analysis is to compute the locations of all program variables at the beginning and end of each basic block. The location of a variable is an element from the lattice $\mathcal{L}$, defined over the set

$$\mathcal{T} = \{Unknown, CPUAndGPU, GPU, CPU\}$$

with the partial order $\prec$ and the join operator $\wedge$ where,

$$\bot = Unknown, \top = CPU$$

$$Unknown \prec CPUAndGPU \prec GPU \prec CPU$$

The reason CPU dominates GPU in the lattice is that the CPU is the coordinating processor. At a program point where a variable is on the GPU along one reaching path, and on the CPU on another, it is probably better to move the variable to the CPU. Also, $CPUAndGPU$ is dominated by both CPU and GPU to avoid extra transfers. Consider the case when GPU is dominated by $CPUandGPU$. Then, at a program point where a variable is on the GPU on one reaching path, and on both along another, a transfer would have been needed for the GPU path to make the variable available on both. It is faster to just say that the variable is no longer available on the CPU. Similar reasoning applies for the CPU case.

If $Vars$ is the set of all variables in the program and B is the set of all basic blocks, for each basic block $b \in B$ and each $v \in Vars$, the following values are defined.

1. $in(v, b) \in \mathcal{T}$ is the location of v at the start of b.
2. $out(v, b) \in \mathcal{T}$ is the location of v at the exit of b.
3. $asmp(v, b) \in \mathcal{T}$ is the mapping of all statements in b that read v before it is written in b. It is $\bot$ if v is not accessed in b.
4. $grnt(v, b) \in \mathcal{T}$ indicates where v will be after the execution of b. If v is not used in b it is defined to be $\bot$.

The data flow analysis is defined by the following equations.

$$in(v, b) = \begin{cases} asmp(v, b) & \text{if } asmp(v, b) \neq \bot; \\ \bigwedge_{b' \in preds(b)} out(v, b') & \text{Otherwise.} \end{cases} \tag{5}$$

$$out(v, b) = \begin{cases} grnt(v, b) & \text{if } grnt(v, b) \neq \bot; \\ in(v, b) & \text{Otherwise.} \end{cases} \tag{6}$$

A standard forward data flow analysis algorithm [1] is used to find a fixed point for $in(v, b)$ and $out(v, b)$ using the data flow equations defined in equations 5 and 6. When a fixed point is reached for in and out, the values of $in(v, b)$ and $out(v, b)$ signify where v needs to be at the start of basic block b and where v will be at the end of basic block b.

The motivating example has three basic blocks as shown in Figure 3. The values of in and out for the variable A are: $in(A, b_1) = Unknown$; $out(A, b_1) = CPU$; $in(A, b_2) = GPU$; $out(A, b_2) = GPU$; $in(A, b_3) = CPU$ and $out(A, b_3) = CPU$.

4.4.2 Edge Splitting

Once the data flow analysis is performed, the location of every variable at the entry and exit of every basic block is known. When a variable changes locations across a control flow edge, a transfer needs to be inserted. More precisely, a transfer for a variable v needs to be inserted on a control flow edge $e = (b_1, b_2)$, when $out(v, b_1) \neq in(v, b_2)$.

MEGHA does not have the ability to explicitly split a control flow edge as it emits structured C++ code[8]. We therefore split edges by dynamically determining which control flow edge was taken to reach the current basic block. This is done by assigning ID numbers to each basic block and inserting code to set the value of a variable, _prevBasicBlock, to the ID at the end of the basic block. Then, at the entry of basic blocks, _prevBasicBlock can be checked to determine which control flow edge was taken and the required transfers can be performed. For the example in Figure 3, the edges (b_1, b_2) and (b_2, b_3) need to be split. This results in the following code.

```
n = 100;
A = rand(n, n);
_prevBasicBlock = 1;
for i=1:n
   if (_prevBasicBlock == 1)
      transferToGPU(A);
   A = A * A;
   _prevBasicBlock = 2;
end
if (_prevBasicBlock == 2)
   transferToCPU(A);
print(A);
```

4.5 Code Generation

Currently our code generator generates C++ code for parts of the input program mapped to the CPU and CUDA kernels for all kernels mapped to the GPU. For kernels that are assigned to the CPU, the loop nests are generated with the computed CPU loop order.

For a kernel mapped to the GPU, each thread of the generated CUDA kernel performs the computation for a single iteration of the surrounding loop nest. Assignment of threads to points in the iteration space is performed as described in Section 4.2. For the example in Section 4.2, the following CUDA kernel would be generated (Some details related to indexing have been simplified. Sizes of Ey and tempVar2 need to be passed to the kernel to correctly index them on line 13).

```
1: __global__ void
2: gpuKernel(float *Ey, float *tempVar2,
3:           int Sx, int Sy, int Sz)
4: {
5:  int tidx = (blockIdx.x*blockDim.x)+threadIdx.x;
6:  int tidy = (blockIdx.y*blockDim.y)+threadIdx.y;
7:  j = tidx;
8:  i = floor(tidy/Sz);
9:  k = tidy % Sz;
10: if (i>Sx || j>Sy || k>Sz)
11:   return;
12: indexVar1 = 2 + (k-1);
13: tempVar2(i, j, k) = Ey(i, j, indexVar1);
14:}
```

The above kernel would be called as follows.

```
dim3 grid(ceil(Sx/32), Sy*Sz, 1);
dim3 block(32, 1, 1);
gpuKernel<<< grid, block >>>(Ey, tempVar2,
                             Sx, Sy, Sz);
```

[8] This decision was made so that the optimizations of the C++ compiler would be more effective.

We chose to generate GPU kernels in this form since it is suitable for further automatic optimization [29]. For kernels that perform matrix multiplication, the code generator emits calls to the matrix multiply in CUBLAS rather than trying to generate an optimized matrix multiply kernel. This is possible because a matrix multiply kernel cannot be composed with other kernels (Section 4.1).

To perform memory transfers, the code generator uses the blocking cudaMemcpy routine. Therefore, the generated code does not overlap memory transfers with computation. However, it is possible to implement a code generation scheme that achieves this. Such a scheme would further improve the performance of the generated code. We leave this to future work.

5. Experimental Evaluation

We implemented MEGHA using the GNU Octave [11] system and used the benchmarks listed in Table 1 to evaluate it. These benchmarks are from several previous projects[9] with the exception of bscholes, which was developed in-house. For each benchmark the table shows the number of tasks identified by our compiler, e.g. clos has a total of 37 tasks out of which 27 are GPU friendly kernels and 10 are CPU only tasks. The number of kernels can be larger than the number of MATLAB statements as the compiler frontend generates many IR statements for each statement in the MATLAB code. We had to make minor modifications to capr as our compiler does not support *auto-growing arrays*. We modified the loop counts to increase the amount of computation in clos and fiff. As our compiler does not support user functions, we manually inlined calls to user-defined functions in all these benchmarks. The code generated by our compiler was compiled using nvcc (version 2.3) with the optimization level -O3. The generated executables were run on a machine with an Intel Xeon 2.83GHz quad-core processor[10] and a GeForce 8800 GTS 512 graphics processor. We also ran these on a machine with the same CPU but with a Tesla S1070 [20]. Both machines run Linux with kernel version 2.6.26. We used the time system utility to measure the runtimes of the executables generated by MEGHA and used the MATLAB function clock to measure the runtime for MATLAB programs run in the MATLAB environment.

For each benchmark, we report the runtime in the MATLAB environment, execution time of CPU only C++ code generated by our compiler and the execution time of C++ plus CUDA code generated by our compiler and the speedups achieved by these two versions over MATLAB execution. Each benchmark was run multiple times. The reported runtime is the average execution time per run. Also, all experiments were performed in single precision since the GeForce 8800 does not have support for double precision arithmetic.

Table 2 shows the runtimes for benchmarks with data parallel regions. In benchmark bscholes we get a speedup of 191X over MATLAB for large problem sizes. In this benchmark our compiler was able to identify the entire computation as one kernel and execute it on the GPU. MATLAB performs better than our CPU only code in bscholes as it handles full array operations better than our C++ code. Speedups for fdtd are higher on Tesla (63X-94X) as memory accesses are coalesced better on the Tesla than on the 8800. We did not report runtimes for the benchmark clos with C++ code as it uses matrix multiplication and our C++ code generator at present generates naïve code for matrix multiplication. However, these routines can easily be replaced by

[9] http://www.ece.northwestern.edu/cpdc/pjoisha/MAT2C/

[10] Although our experimental system contained two quad-core Xeon processors (8 cores), in our experiments, we use only a single core as the present compiler schedules tasks to a single GPU and a single CPU core. The compiler can easily be extended to use multiple CPU cores.

Benchmark	Size	Execution Time (in Seconds)				Speedup over MATLAB		
		MATLAB	CPU	CPU+8800	CPU+Tesla	CPU	CPU+8800	CPU+Tesla
bscholes	51200 Options, 100 Iters	0.9	1.99	0.03	0.02	0.45	30	45
	102400 Options, 500 Iters	8.9	19.7	0.07	0.05	0.45	127	178
	204800 Options, 500 Iters	17.2	39.5	0.1	0.09	0.43	172	191
clos	1024 vertices	0.5	-	0.18	0.11	-	2.8	4.5
	2048 vertices	4.1	-	1.13	0.73	-	3.6	5.6
	4096 vertices	30.8	-	mem-exc	5.82	-	-	5.3
fdtd	512 x 512 x 15	187	13.5	10.1	2.94	13.8	18.5	63.6
	512 x 512 x 32	408	30.5	22.69	5.48	13.4	18.0	74.4
	1024 x 512 x 32	806	62.7	mem-exc	9.8	12.8	-	82.2
	1024 x 1024 x 32	1623	121.6	mem-exc	17.2	13.3	-	94.4
nb1d	4096 bodies	14.8	7.6	5.4	5.4	1.95	2.7	2.7
	8192 bodies	50.1	29.2	13.7	15	1.7	3.6	3.3
	16384 bodies	513	323	119	128	1.6	4.3	4.0
nb3d	512 bodies	42.3	12.4	4.82	3.64	3.4	8.8	11.6
	1024 bodies	224.8	47.5	17.8	16.7	4.7	12.6	13.5
	1536 bodies	508	101.5	38.6	37.15	5.0	13.2	13.7

Table 2: Runtimes and speedups for data parallel benchmarks

Benchmark	Description	# of Lines	Tasks
`bscholes`(dp)	Stock Option Pricing	50	35/7
`capr`	Line Capacitance	60	69/23
`clos`(dp)	Transitive Closure	30	27/10
`crni`	Heat Equation Solver	60	55/14
`dich`	Laplace Equation Solver	55	68/15
`edit`	Edit Distance	60	35/15
`fdtd`(dp)	EM Field Computation	70	74/12
`fiff`	Wave Equation Solver	40	45/11
`nb1d`(dp)	1D N-Body Simulation	80	63/18
`nb3d`(dp)	3D N-Body Simulation	75	69/26

Table 1: Description of the benchmark suite. (dp) indicates benchmarks with data parallel regions. Number of kernels/number of CPU only statements are reported under the column Tasks.

calls to optimized BLAS libraries. The GPU versions for `clos`, which use the CUBLAS library, achieve a speedup[11] of 2.8X-5.6X on 8800 and Tesla. For `nb1d` and `nb3d` the GPU versions achieve speedups of 2.7X-4.0X and 8.8X-13.7X respectively. Lastly, even the CPU only version gives considerable speedups of 1.6X-13.4X for `fdtd`, `nb1d` and `nb3d` on a single CPU core.

Table 3 shows the runtimes for benchmarks with no or very few data parallel regions. The performance benefits over MATLAB are mainly due to compiling the MATLAB code to C++. Even here, we observe an improvement in execution time by factors of 2X-10X.

Table 4 compares performance of code generated by our compiler with that of GPUmat. GPUmat requires the user to explicitly mark variables as GPU variables before an operation on them can be performed on the GPU. The user has to manually identify data parallel regions and insert code to move data to the GPU only in these regions. Incorrect use of GPU variables can have an adverse impact on performance. We used some information from the code generated by our compiler to mark the GPU variables. GPUmat performs considerably better than MATLAB. However, because our compiler can automatically identify GPU kernels and reduce the amount of data transfer by intelligent kernel composition, we were able to get better performance (1.5X-10X) than GPUmat.

[11] Benchmarks `clos` (with 4096 vertices) and `fdtd` (problem size of 1024x512x32 or 1024x1024x32) could not be run on the 8800 as the kernel memory requirement exceeded the available memory (512MB) in the GPU.

Benchmark	Size	Execution Time (in Seconds)		
		MATLAB	CPU	CPU+8800
capr	256 x 512	9.15	1.43	1.43
	500 x 1000	17.3	2.65	2.66
	1000 x 2000	34.7	5.37	5.38
crni	2000 x 1000	16.05	6.33	6.39
	4000 x 2000	64.1	24.7	23
	8000 x 4000	254.8	104	103
dich	300 x 300	1.24	0.3	0.3
	2000 x 2000	55.7	8.01	8.15
	4000 x 4000	249.2	31.95	31.89
edit	512	0.35	0.11	0.11
	4096	22.1	0.45	0.45
	8192	88.1	1.47	1.48
fiff	450 x 450	0.51	0.22	0.21
	2000 x 2000	2.04	0.41	0.41
	4096 x 4096	8.17	2.5	2.51

Table 3: Runtimes for non data parallel benchmarks

Benchmark	Size	Execution Time (in Seconds)		
		MATLAB	GPUmat	CPU+Tesla
`bscholes`	102400, 500	8.9	7.2	0.08
`clos`	2048	4.1	1.13	0.73
`fdtd`	512 x 512 x 15	187	30	2.94
`nb1d`	4096	14.8	12.03	5.44
`nb3d`	1024	224.8	172	16.67

Table 4: Comparison with GPUmat

Our compiler reorders loops to improve cache locality on CPUs and coalesced accesses on GPUs. Table 5 shows the performance gains obtained by reordering loop nests. The benchmark `fdtd` has 3 dimensional arrays and benefits a lot on both the CPU and GPU with this optimization. Loop reordering improved the performance of `fdtd` CPU+GPU code by a factor of 16 and CPU only code by a factor of 5.1. Whereas for `nb3d` this optimization results in an improvement of 3X for CPU+GPU code and 1.05X for CPU only code.

Benchmark	Execution Time (in Seconds)				
	MATLAB	CPU Only		CPU + GPU	
		No Opt	Opt	No Opt	Opt
`fdtd`	187	82.05	15.82	48.14	2.94
`nb3d`	224.8	50.1	47.5	47.6	16.7

Table 5: Performance gains with parallel loop reordering. Problem sizes are 512 x 512 x 15(`fdtd`) and 1024(`nb3d`).

6. Related Work

Several previous projects have studied the problem of compiling MATLAB. The first of these was the FALCON project that compiled MATLAB programs to FORTRAN [10]. Most of our frontend is based on work done in the FALCON project. MaJIC [2] is a just-in-time MATLAB compiler that compiled and executed MATLAB programs with a mixture of compile time and run-time techniques. The MAGICA project [16] focussed on statically inferring symbolic types for variables in a MATLAB program and establishing relationships between shapes of various variables. Mat2C [17] is a MATLAB to C compiler based on MAGICA. MATCH [13] is a virtual machine that executes MATLAB programs in parallel by mimicking superscalar processors. The possibility of using expensive compile time analysis for generating high performance libraries from MATLAB routines was studied by Chauhan et al [7]. More recently, McVM [8], a virtual machine for the execution of MATLAB programs, that performs specialization based on types has been developed. However, none of these projects studied automatically compiling MATLAB to machines with distributed memory or heterogeneous processors.

Jacket [14] and GPUMat [12] are systems that enable users to execute parts of MATLAB programs on GPUs. However, these require users to specify what is to be run on the GPU by annotating the source code. Arrays whose operations are to be performed on the GPU are declared with a special type. This is similar to the concept of data parallel arrays in Accelerator [25]. Similarly, Khoury et al extend Octave to target the Cell BE processor [19]. However, their system still requires users to cast matrices to a specific type so that operations on these matrices can be performed on the SPEs. It also uses a virtual machine based approach to perform matrix operations on the SPEs as opposed to generating custom code.

Several previous works have studied problems related to compilation for GPUs. Techniques for optimizing naïve CUDA kernels were studied by Yang et al [29]. The use of the polyhedral model to optimize code for accelerator architectures was studied by Baskaran et al [4][5]. These techniques are complementary to those discussed in this paper as they can be used to optimize kernels and low level code generated by our compiler.

Previous works have also explored compiling programs in other high level languages to GPUs. For example, compilation of stream programs to GPUs and heterogeneous machines was studied by Udupa et al [26][27]. To the best of our knowledge, our paper is the first to study the problem of automatically compiling MATLAB programs for execution on systems with heterogeneous processors.

7. Conclusions and Future Work

In this paper, we presented a compiler framework, MEGHA, for the automatic compilation of MATLAB programs for execution on systems with heterogeneous processors. We presented a technique for identifying and composing kernels in the input MATLAB program while taking several costs and benefits into account. An efficient heuristic for automatically mapping and scheduling identified kernels on CPU and GPU has also been proposed. The heuristic takes into account data transfer while mapping kernels to processors. Then, a data flow analysis based edge splitting strategy to handle dependencies across basic blocks was proposed. The proposed framework was implemented and initial performance results indicate that our approach is promising with performance gains up to 191X over native MATLAB execution being obtained.

There are also several interesting directions for future work. One interesting direction is to implement the techniques described in this paper as a mixture of static and dynamic optimizations. Also, in this paper, we did not consider GPU memory constraints while mapping and scheduling kernels. This needs to be considered for larger programs to be correctly run. Optimizing the generated kernels using shared memory and performing other optimizations on them also needs to be studied.

Acknowledgments

We thank R. Manikantan and other members of the Lab for High Performance Computing, IISc for several helpful discussions regarding the work described in this paper. We would also like to thank the anonymous reviewers for several valuable suggestions that helped improve our paper. We would also like to acknowledge funding received from Microsoft and NVIDIA.

References

[1] A. V. Aho, Ravi Sethi, J. D. Ullman, M. S. Lam. *Compilers: Principles, Techniques, & Tools*. Pearson Education, 2009.

[2] G. Almasi, D. Padua. *MaJIC: Compiling MATLAB for Speed and Responsiveness*. In the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02).

[3] ATI Technologies, *http://ati.amd.com/products/index.html*

[4] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan. *A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs*. In the 22nd Annual International Conference on Supercomputing (ICS '08).

[5] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan. *Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories*. In the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08).

[6] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan. *A Practical Automatic Polyhedral Parallelizer and Locality Optimizer*. In the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI '08).

[7] A. Chauhan, C. McCosh, K. Kennedy, and R. Hanson. *Automatic Type-Driven Library Generation for Telescoping Languages*. In the 2003 ACM/IEEE Conference on Supercomputing (SC '03).

[8] M. Chevalier-Boisvert, L. Hendren, C. Verbrugge. *Optimizing MATLAB Through Just-In-Time Specialization*. In the 2010 International Conference on Compiler Construction (CC '10).

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. In the ACM Transactions on Programming Languages and Systems, 13(4):451–490, Oct. 1991.

[10] L. De Rose, D. Padua. *Techniques for the translation of MATLAB programs into Fortran 90*. In the ACM Transactions on Programming Languages and Systems, 21(2):286–323, Mar. 1999.

[11] J. W. Eaton. *GNU Octave Manual*, Network Theory Limited, 2002.

[12] GPUMat Home Page. *http://gp-you.org/*

[13] M. Haldar et. al. *MATCH Virtual Machine: An Adaptive Run-Time System to Execute MATLAB in Parallel*. In the 2000 International Conference on Parallel Processing (ICPP '00).

[14] Jacket Home Page. *http://www.accelereyes.com/*

[15] P. Joisha, P. Banerjee. *Static Array Storage Optimization in MATLAB*. In the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03).

[16] P. Joisha, P. Banerjee. *An Algebraic Array Shape Inference System for MATLAB*. ACM Transactions on Programming Languages and Systems, 28(5):848–907, September 2006.

[17] P. Joisha, P. Banerjee. *A Translator System for the MATLAB Language*, Research Articles on Software Practices and Experience '07.

[18] K. Kennedy, K. S. McKinley. *Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution*. In the 6th International Workshop on Languages and Compilers for Parallel Computing (LCPC '93).

[19] R. Khoury, B. Burgstaller, B. Scholz, *Accelerating the Execution of Matrix Languages on the Cell Broadband Engine Architecture*. IEEE Transactions on Parallel and Distributed Systems, 22(1):7–21, Jan. 2011.

[20] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym. *NVIDIA Tesla: A Unified Graphics and Computing Architecture*. IEEE Micro, March 2008.

[21] Mathworks Home Page. *http://www.mathworks.com/*

[22] NVIDIA Corp, *NVIDIA CUDA: Compute Unified Device Architecture: Programming Guide*, Version 3.0, 2010.

[23] NVIDIA Corp, Fermi Home Page, *http://www.nvidia.com/object/fermi_architecture.html*

[24] S. K. Singhai, K. S. Mckinley. *A Parametrized Loop Fusion Algorithm for Improving Parallelism and Cache Locality*, Computer Journal, 1997.

[25] D. Tarditi, S. Puri, J. Oglesby. *Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses*. In the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII).

[26] A. Udupa, R. Govindarajan, M. J. Thazhuthaveetil. *Software Pipelined Execution of Stream Programs on GPUs*. In the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09).

[27] A. Udupa, R. Govindarajan, M. J. Thazhuthaveetil. *Synergistic Execution of Stream Programs on Multicores with Accelerators*. In the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '09).

[28] V. Volkov, J. W. Demmel. *Benchmarking GPUs to Tune Dense Linear Algebra*. In the 2008 ACM/IEEE Conference on Supercomputing (SC '08).

[29] Y. Yang, P. Xiang, J. Kong, H. Zhou. *A GPGPU Compiler for Memory Optimization and Parallelism Management*. In the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '10).