PLASMA: Portable Programming for SIMD Heterogeneous Accelerators

Sreepathi Pai, R. Govindarajan, M. J. Thazhuthaveetil
Supercomputer Education and Research Centre,
Indian Institute of Science.
Email: {sree@hpc.serc, govind@serc, mjt@serc}.iisc.ernet.in

***Abstract*—Data-parallel accelerators have emerged as high-performance alternatives to general-purpose processors for many applications. The Cell BE, GPUs from NVIDIA and ATI, and the like can outperform conventional superscalar architectures, but only for applications that can take advantage of these accelerators' SIMD architectures, large number of cores, and local memories. Coupled with the SIMD extensions on general-purpose processors, these heterogeneous computing architectures provide a powerful platform to accelerate data-parallel programs. Unfortunately, each accelerator provides its own programming model, and programmers are often forced to confront issues of distributed memory, multithreading, load-balancing and computation scheduling. This necessitates a framework which can exploit different types of parallelism across heterogeneous functional units and supports multiple types of high-level programming languages including stream programming or traditional shared or distributed memory programming framework or prototyping languages such as MATLAB.**

Towards this goal, in this paper, we present PLASMA, a programming framework that enables the writing of portable SIMD programs. The main component of PLASMA is an intermediate representation (IR), which provides succinct and clean abstractions to enable programs to be compiled to different accelerators. With the assistance of a runtime, these programs can then be automatically multithreaded, run on multiple heterogeneous accelerators transparently and are oblivious of distributed memory. We demonstrate a prototype compiler and runtime that targets PLASMA programs to scalar processors, processors with SIMD extensions and GPUs.

I. Introduction

Modern high-performance computing systems use a combination of general-purpose homogeneous superscalar cores and specialised accelerators for performance. Such systems are characterised by a large number of processing elements and heterogeneity in cores. For example, accelerators such as the Cell BE [1], Clearspeed [2], and Graphics Processing Units (GPUs) [3] all exhibit these characteristics. Modern GPUs provide sixteen or more processors that are internally organised as a number of SIMD cores [3]. Programming these large cores is further complicated by the fact that programmer needs to manage memory explicitly so as to extract higher memory performance. For example, programs on the GPU routinely consist of thousands of threads but can only access data stored in device memory. The Cell BE, on the other hand, provides eight SIMD cores [1] but unfortunately limits each core to a 256KB local store. Despite these restrictions, for data-parallel programs that can utilise the raw compute power of these accelerators, speedups achieved are very large.

The proliferation of specialised accelerators, unfortunately, has not been accompanied by ease in programming them. The quirks of individual accelerators, their distributed memory architecture, the large amount of data-level parallelism required to amortise their use, and the considerable heterogeneity exhibited by these processors have all made programming them individually a major challenge, to say nothing of the complexity of programming a system which contains multiple accelerators.

Current approaches are limited and follow an *ad hoc* approach in exploiting some aspect of the parallelism. Each accelerator exposes a different programming model with most relying on one or more of (i) auto-vectorisation capabilities of the platform compiler [4], (ii) use of language and architecture-specific "intrinsics" [5], (iii) compiler extensions (such as GCC 4.x vector attribute of a type), (iv) vendor-specific programming models such as NVIDIA's CUDA [6] or ATI's CAL, (v) vendor-agnostic language extensions such as OpenCL [7].

Each of these approaches have their own limitations. Almost all require the programmer to manually write multithreaded code. Some do not allow variable-length vectors. Those that are portable limit themselves to a subset of SIMD instructions provided by the platform. Vendor-specific programming models such as CUDA and CAL require the programmer to climb a fairly steep learning curve. They also do not allow portability across different platforms. Further, the host CPU is often underutilised by these accelerator models. Even the recently proposed OpenCL [7] language extensions for programming heterogeneous systems are not very high level and continues to expose many of these architecture-specific issues to the OpenCL programmer.

Programmers can resort to a higher-level data-parallel programming model, like Stream programming [8], [9]. But stream programming is more suited for streaming data; further *implementors* of compilers and runtimes for these stream programming models will still have to wrestle with all of the issues imposed by the heterogeneity of the devices they're targeting.

The requirements call for a single common framework designed to exploit different types of parallelism across heterogeneous functional units, or portable across different generations of the same processor/accelerator, or different processors. Not only should such a framework be retargetable

to all these hardware accelerator architectures, it should also function as a target for other higher-level languages as well. It should, for example, be possible to compile stream programs written in Brook, StreamIT or LABVIEW or OpenMP or MPI programs or even programs written in array languages such as MATLAB to compile to this framework. This would allow the large existing legacy codebase to benefit from accelerators. Designing such a high-level programming framework which allows multiple high-level languages to be compiled for several targets with vastly different architecture features requires an intermediate representation which has abstracted the underlying architecture features and different types of parallelism in a clean and succinct manner, yet is powerful enough to expose and exploit the architectural features to extract maximum performance.

Towards this goal, in this paper, we present PLASMA, a programming framework for heterogeneous multicore systems that allows programmers to write portable code for these devices. The PLASMA framework primarily provides an IR which cleanly abstracts any particular SIMD implementation and also permits efficient translation to other SIMD processors. With the assistance of a runtime, programs compiled into PLASMA can be automatically threaded, can run on different devices simultaneously and are oblivious of multiple address spaces. The framework is extensible and high-level programming languages, including stream programming languages, can be compiled into the PLASMA IR representation.

In this paper, we make the following contributions:

- We present an architecture and language-neutral portable intermediate representation for SIMD data-parallel programs.
- We demonstrate compilation techniques for generating efficient code from this IR for multicore CPUs with SIMD extensions and the NVIDIA family of GPUs, specifically the NVIDIA 8800 and the Tesla S1070.
- We develop a runtime system that allows these portable programs to run on heterogeneous systems utilising all processing elements on all devices.
- Our results show that the performance of portable PLASMA programs is competitive to platform-specific optimised code.

In the next section, we present an overview, the PLASMA framework and its components. The design of the PLASMA IR, its constructs and their semantics are described in Section III. We describe our compiler and runtime implementation in Section IV. Performance results from a prototype compiler and runtime system that targets scalar processors, the SSE3 vector extensions and CUDA are presented in Section V. We compare our work to other SIMD abstractions, similar data parallel frameworks and platform compilers in Section VI.

II. PLASMA Overview

As mentioned in the introduction, our aim is to design a single common programming framework which can exploit different types of parallelism across heterogeneous functional units, and portable across different generations of the same processor/accelerator, as well as different processors. Towards this goal we propose PLASMA, a programming framework for heterogeneous multicore systems which allows programmers to write portable code.

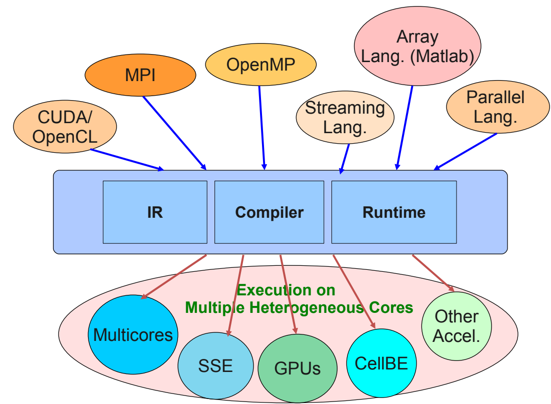

Fig. 1. PLASMA Overview

An overview of the programming framework is presented in Figure 1. The framework is intended for programming in a high-level stream programming language, such as StreamIt [8], Brook [9], OpenMP, MPI or MATLAB which can be compiled to the PLASMA IR. Users can also directly write their programs in the PLASMA IR, which is a high-level language in itself.

The PLASMA framework consists of (i) a set of abstractions for the SIMD programming model, (ii) an IR that embodies those abstractions, (iii) a compiler for the IR, and (iv) a run-time system (refer to Figure 1). The compiler and the run-time system form a PLASMA implementation. The abstractions and the IR ensure that any PLASMA program can be compiled to any device (CPU, GPU, etc.) that is supported by a PLASMA implementation.

Our implementation currently takes a single PLASMA source file, written independently of any SIMD device, and compiles it to programs for single-core CPUs, multi-core CPUs and the NVIDIA GPUs (CUDA). CPU-supported vector extensions, such as SSE, are enabled as well. Threading is done automatically for multi-core CPUs. Further, the runtime system takes care of distributed memory and ensures that data is moved appropriately between host (CPU) memory and device (GPU) memory, before the computation is performed.

The salient features of the PLASMA framework are:

PLASMA IR can be the target of many high-level languages. It should be noted that the PLASMA framework is merely *our* implementation. Our IR is designed to be reusable across any project that targets SIMD devices. For example, writers of stream programming languages/frameworks (like Brook, StreamIt, etc.) can target our IR and gain automatic access to all the devices supported by PLASMA. Since the IR is "minimal" – it does not dictate, for example, scheduling policy – it does not affect the higher level goals of stream programming, which is to encapsulate pipeline parallelism, but does save the significant effort involved in targeting each

device separately.

PLASMA IR can be targeted for device-specific optimizations. PLASMA code is also highly portable across devices. A serialised version of our IR enables the *same* code to run efficiently across a wide range of devices – from embedded ARM Neon-based [10] devices to massively-multicore multithreaded workstations. On each of these devices, the code can take full advantage of all device-specific features. This flexibility comes from the architecture-independent design of our IR and not from conforming to lowest common denominator features (or over general features) of target devices. For example, a CPU backend need not concern itself with the distributed memory issues that confront a GPU or Cell BE backend. As each backend can be written by an expert for that particular device, PLASMA compilers can generate highly efficient code that can take full advantage of present and future hardware.

PLASMA can reuse existing concepts/features. Representing computation is not the only problem confronted by parallel programmers. They must also deal with issues like load-balancing and scheduling. In PLASMA, these are part of the run-time system and as such can be replaced as necessary. Our current implementation uses a simple RPC-stub like mechanism with static work partitioning to interface to different accelerators, but a mechanism like work queues could also be used to implement a dynamic work sharing scheme across accelerators.

In PLASMA, data-parallel types are first class citizens. By treating a data-parallel type as a first-class member of the IR, it is easier to perform several device-independent and device-specific optimizations. Register blocking is one example of the former, while easily and correctly taking advantage of non-temporal load instructions is an example of the latter.

In this paper, we discuss the abstractions that govern our design of the IR. We then describe the backend and run-time system of a PLASMA IR compiler that demonstrates writing and compiling single-source programs for heterogeneous accelerators.

III. PLASMA IR

The PLASMA intermediate representation (IR) for data-parallel SIMD programs is designed to abstract away the details of any particular SIMD architecture. Each instruction in the IR is composed from four constructs: the Vector datatype, Operators, Distributors and Vector compositions. This representation allows us to be independent of both a high-level language (which may have its own data parallel constructs) as well as a specific architecture.

VCODE [11] is one of the first IRs proposed to encode data-parallel programs. It was designed to abstract the many high-level languages that were prevalent at that time. More recently, Vector LLVA [12] was proposed as a means of abstracting the many architectures that support SIMD processing while still yielding good performance. However, neither was designed nor demonstrated to work on multiple heterogeneous cores or deal with distributed memory. For example, Vector LLVA supports programs that use fixed-length vectors *or* variable-length vectors but not both. While this works across similar accelerators (e.g., from SSE to AltiVec), it doesn't produce efficient code when going from, say, SSE to the GPU.

We therefore describe our proposal for a portable IR that addresses these architectures by first highlighting some requirements that any such IR would have to handle.

A. Requirements of an IR for Heterogeneous Systems

In order to effectively exploit the parallelism and heterogeneity supported by various accelerators, any intermediate representation should have the following capabilities.

Ability to abstract data parallelism granularity The parallelism supported by different SIMD architectures varies widely. Most general-purpose processors today support parallel operations over "short" vectors that range from 128-bits to 256-bits [5]. However, most GPUs [3] today support processing over *hundreds* of 32-bit elements. Therefore, to keep all the processing elements busy, an IR must encode data elements in a "bulk" data type whose length depends only on the application/algorithm and not on architectural parameters.

Ability to abstract vector instructions General-purpose CPUs support short-vector processing via SIMD instruction set extensions which are mostly vector extensions of scalar instructions. However there are also a small but significant number of "true vector" instructions like AltiVec's `vpermute` [5] which have no direct primitive scalar equivalent. An IR must support both these types of instructions without restricting vector instructions like `vpermute` to operate only on the entire bulk data type. That is, it must be possible to perform `vpermutes` in parallel on a bulk data type. Doing so also allows such an IR to naturally support the many application-specific instructions endemic to such instruction set extensions (e.g., the `sad` instruction performing the sum of absolute differences used in the motion estimation phase of a video codec).

Ability to abstract common parallel control idioms Control structures like the familiar `map` and `reduce` are widely used high-level parallel idioms that have varying implementations across different accelerators. They should therefore be preserved as high-level abstractions in the IR as opposed to encoding them using primitive control constructs. This has four major benefits. First, it allows the use of a complex frontend compiler, e.g., a vectorizing compiler producing PLASMA IR, to detect these idioms in case they are not explicitly specified. Second, the IR compiler can parameterize these structures at runtime for the parallelism that is available in the hardware. Third, other control idioms like the Scan [13] – which was developed for use on vector architectures and have recently also found use on GPUs [14] – can be deployed similarly facilitating easy implementation and portability to other architectures. Finally, the compiler for the IR can trivially take advantage of hardware assists for these idioms, e.g., using the BlueGene's collective network [15] to run a `reduce`.

Ability to specify data access patterns An IR that targets devices with private and distinct address spaces must minimize data transfers between them. This can be hard to do efficiently if data accesses cannot be specified at a fine enough granularity, since algorithms will sometimes operate only on a subset of the elements of a bulk data type. Consider the case of motion detection kernels in most video codecs which often operate only on a small window of a much larger frame at a time. They may force the runtime to end up transferring the whole frame. Therefore, an IR must attempt to encode bulk data access patterns at as fine a granularity as possible.

We now describe the PLASMA IR.

B. IR constructs

1) Operators: Operators abstract architectural instructions in our IR. We distinguish between operators that accept Elements, e.g., an integer (or another primitive data type) and those that accept Blocks, i.e., an aggregate of primitive types. The former encodes instructions like `add`, `mul`, etc., while the latter encodes true vector instructions like `vpermute`.

The set of all operators forms the instruction set for the PLASMA IR. Operators are completely specified by their functional semantics, their return type (Element or Block) and number of their arguments:

$$result = op(arg_1, arg_2, \ldots, arg_n)$$

The following examples shows three possible combinations: an element-wise operator `add`, a block operator `permute`, and an operator that returns an element from a block argument `max`:

```
Element sum =  add(Element A, Element B)
Block V_permuted = permute(Block V,
                           Block Perm)
Element V_max = max(Block V)
```

Note that Element is technically a Block of unit length. Currently our implementation supports the same primitive types as C (`char`, `int16_t`, `int32_t`, etc.).

2) Vectors: Vector is the bulk data-parallel type in PLASMA. It is defined as an aggregate of primitive types. Values of this type are created and initialized in an implementation-dependent manner, but each Vector supports a `length` attribute, and can be indexed. The length of a Vector V is denoted by $|V|$. Unless otherwise stated, further references to "vector" in the text refer to the PLASMA Vector type.

An *index vector* is a vector of an primitive type wide enough to hold indices of the largest vector possible (e.g., C99's primitive `size_t` type).

Our implementation only supports vectors whose elements are all of the same primitive data type.

3) Distributors: A distributor maps elements of its vector arguments to arguments of a specified operator. This mapping also exposes data-parallelism specific to the semantics of the distributor.

For example[1], the PLASMA `par` distributor maps an operator to "parallel" elements of the provided vector arguments:

```
A = Vector(1, 2, 3, 4)
B = Vector(6, 7, 8, 9)
C = par(add, A, B)
/* result:  C = (7, 9, 11, 15) */
```

Formally, a Distributor accepts arguments of types Operator, Vector or Scalar. Depending on its semantics, a domain decomposition is performed on its Vector arguments and the specified Operator(s) are executed on this decomposition. Scalar arguments are logically expanded to vectors containing the same value. Our current implementation provides the `par` and `reduce` distributors. The `reduce` distributor requires a binary, associative and commutative operator argument as the first argument and a vector argument as the second argument and produces a vector with a single element as the result.

```
/* scalar argument 2 is logically
   expanded to a vector */
C = par(mul, C, 2)
/* result: C = (14, 18, 22, 30) */

D = reduce(add, C)
/* result: D = (84) */
```

To apply an operator which expects a Block argument, vector arguments to a distributor are qualified with a $block(V, n)$:

```
/* perform max on blocks of size 2 */
A = Vector(1, 2, 5, 6, 9, 10)
dest = par(max, block(A, 2))
/* result: dest = (2, 6, 10) */
```

Distributors can support conditional application of operators based on a supplied mask vector. This mask vector would be usually generated as a result of a PLASMA instruction applying a logical operator. However, not all architectures support masks or can emulate them efficiently. In particular most short vector architectures do not support masks, and implementations must either mask out parts of the vector register using bitwise operations or "degrade" to scalar code with conditionals. Both these techniques reduce performance with the first also requiring handling corner cases where masked values might cause errors (e.g., division resulting in a div-by-zero error). Our current implementation does not support masks.

4) Vector Compositions: As noted in the section on Vectors, the initial creation of a vector is implementation dependent. However, the IR supplies a number of standard *vector compositions* that create a new vector from the contents of existing vectors. These compositions can be used wherever a Vector argument is expected. Our implementation supports the vector compositions in Table I.

Vector compositions can be composed together to specify increasingly complex element-level accesses to a Vector. Such

[1]In the following examples, any variable whose value is not defined in a code segment takes the value defined in the previous examples.

TABLE I
VECTOR COMPOSITIONS (NON-EXHAUSTIVE)

Name	Example
Concatenate	$V_1 + V_2 + \cdots + V_n$
Slice	$V[start : end : stride]$
Reorder	$V[V_{perm}$
Gather	$[V]$ (as source)
Scatter	$[V]$ (as result)

accesses are abstracted from the distributor (which only sees the final result vector) and are also available for analysis to the compiler.

Since most vector compositions appear in anonymous arguments, our implementation takes the liberty of never realizing many of these vectors – it compiles vector compositions to direct accesses of the base vector.

C. Instructions and the Execution Model

By combining each of the above constructs we obtain an instruction in the IR. The bulk operations encoded by these instructions are conceptually executed in program order.

The PLASMA IR does not stand alone and requires an existing scalar ISA to specify control flow, delineate subroutines, etc. In this sense, it is used in an analogous manner to existing short-vector extensions.

D. Encoding Tasks, Stream and SPMD Data-parallelism

When generating code for multiple cores, our current compiler generates task boundaries automatically with a task being an independent thread of control. However, stream languages and SPMD programs have different notions of tasks, boundaries of which are often explicitly specified by the programmer. For StreamIt, this is the "filter", while for SPMD it is the whole program. We are exploring constructs to represent these directly in the IR. Currently, when such explicit tasks exist, the compiler for the PLASMA IR can be instructed to generate code that only exploits the parallelism that is left over (e.g. short-vector).

IV. COMPILATION

We have currently implemented a source-to-source compiler that accepts source files in C with extensions called CPLASM. This dialect introduces a vector data type and simple syntax to allow the embedding of PLASMA instructions in C programs. The output of this compiler consists of a main C file, several architecture-specific files, and files containing stub routines that link the architecture-specific variants of PLASMA code with the main program. The compiler currently has three targets: plain C code, code that uses the SSE3 extensions and CUDA code. We use the OpenMP directives to generate threaded code for the CPU. The compiler can also be used to produce code that uses only a specific accelerator.

In this implementation, a Vector is a contiguous section of memory that has been allocated at an address guaranteed to be aligned with respect to the strictest requirements – currently the GPU, which requires 128-byte alignment.

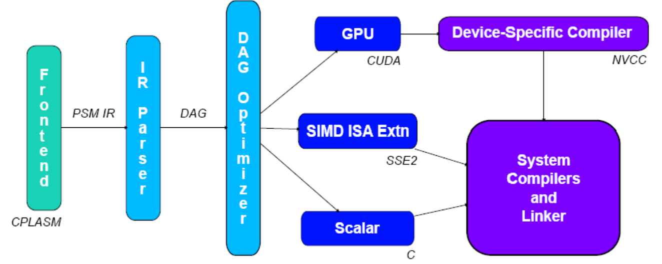

Fig. 2. An overview of the compilation process

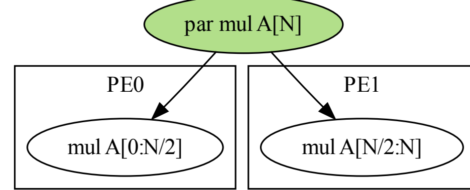

Fig. 3. A single node can distribute its workload across several processing elements.

An alternative implementation also exists that interprets PLASMA instructions at runtime similar to the approach adopted by MS Accelerator [16], RapidMind [17], Sh [18], etc. We do not discuss that implementation here.

A. Architecture-independent Compilation and Optimization

The primary structure we operate upon is the static dataflow DAG of PLASMA instructions. Each node in the DAG represents a PLASMA instruction. All nodes in a single DAG have the same control-dependence. Our primary goal is to form *tasks* from the nodes in a DAG that we can distribute across processing elements. To do this, we first identify all the sources of concurrent tasks in the DAG.

B. Task Formation

1) Tasks from Individual Nodes (TF-ONE): Each individual node is a parallel bulk operation. It can therefore be executed in parallel across all processing elements (see Figure 3). However, this is rarely an efficient method of execution since a single node rarely contains enough work to amortize the code and data transfers involved in executing it on an accelerator like the GPU or the Cell. It may however be used on a shared-memory multicore architecture.

2) Tasks from Multiple Dependent Nodes (TF-DEP): Two strategies exist to exploit parallelism among multiple nodes that have a true dependence. The first involves partitioning the DAG such that each partition executes concurrently on different processing elements (see Figure 4). This exposes the pipeline-parallelism as task-level parallelism and is similar to the Stream programming model.

The second strategy, implemented in our compiler, groups chains of dependences together to form a large "supernode". Since the quantum of work in a supernode is large enough, it may be executed in the fashion of an individual node (TF-ONE, section IV-B1) after domain decomposition.

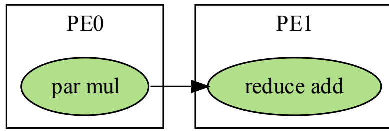

Fig. 4. Dependent nodes can execute across processing elements exposing pipeline parallelism.

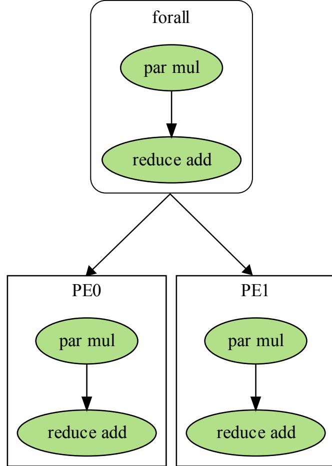

Fig. 5. The iterations of the `forall` loop are split across processing elements

3) Tasks from Nodes in a Loop (TF-LOOP): Nodes in a `while`-loop and a sequential `for`-loop are handled as in Sections IV-B1 and IV-B2. However, nodes in a `forall` loop can also exploit the parallelism exposed by the `forall` loop. For example, consider the SGEMV program:

```
forall Y_i in Y
     par mul, temp, A[row], B
     reduce add, Y_i, temp
end forall
```

In this particular case, it is more profitable to execute each iteration of the `forall` loop across processing elements (see Figure 5) as opposed to executing `par` and then a parallel `reduce`.

4) Tasks from Independent Nodes (TF-IND): DAGs that are independent and have the same control dependence can be executed concurrently across multiple processing elements (see Figure 6). We do not support this in our compiler yet.

C. Optimizations

In this section, we detail architecture-independent optimizations that apply to our IR. In the subsequent section we discuss architecture-specific optimizations.

1) Optimizing for the Memory Hierarchy: The presence of vector code necessitates the use of vector optimizations [19], and our compiler performs the following well-known optimizations:

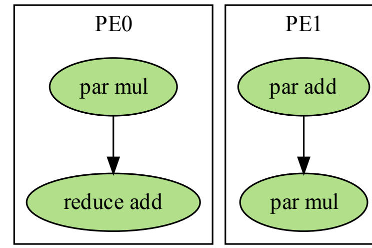

Fig. 6. Independent DAGs execute concurrently across processing elements

- Loop fusion
  The loops of dependent instructions can be fused together to increase memory locality and decrease loop administrative overhead. For example, in the SGEMV benchmark in Section IV-B3, the `par` and `reduce` generate serial loops. These loops have the same iteration count and can therefore be fused into a single loop.
- Blocking/Tiling
  The use of bulk data types can destroy locality between instructions. We therefore support block access to vectors at several levels. For `reduce`, we block at the register level. For both `par` and `reduce` distributors, we block at the memory level, but expect a higher-level compiler to have generated the blocking code.
- Removal of Temporary Vectors
  The compiler attempts to identify temporary vectors that hold intermediate results and eliminates them. While this reduces memory usage, a more significant gain is the reduction in transfer time between address spaces.
  As an example, fusing the `par` and `reduce` loops in the SGEMV example of Section IV-B3 also results in the elimination of a temporary vector.
- Overlapping Computation with Communication
  Our CUDA backend generates CUDA 1.1 code which does not support overlapping computation with memory transfers. However nothing in the IR prohibits such an optimization and we plan to implement this optimization for an accelerator like the Cell (or for CUDA 2.0) which does support asynchronous transfers.

2) Code Movement: Since invoking a remote stub is expensive, the compiler moves the loops that enclose DAGs into the remote stubs. This is currently done for sequential `for` loops and for `forall` loops.

D. Compiling for the GPU

There are two programming models for GPUs. The first model requires that all programs be compiled into shaders that the graphics pipeline can then execute. This model has various limitations including the size of the shaders, the number of inputs and outputs for a shader, prohibition on loops, lack of scatter support and so on. This is the model that most early work [16], [18] that compiles to a GPU has targeted.

In contrast, modern GPUs now support what NVIDIA calls a “compute-unified” model for programming [20]. In this model the GPU is presented as a very wide array of processors [3]. Programs written for this model have none of

the limitations of the shader model. We follow this approach. As such, a larger class of programs can be compiled to the GPU.

1) CUDA Specifics: We refer the reader to [6] for a detailed description of the CUDA programming model. Here, we describe only the basic architecture and memory hierarchy.

In CUDA, each GPU is composed of many multiprocessors. Each multiprocessor contains multiple scalar processors. All of the scalar processors in a multiprocessor share the same instruction stream, though some may be masked off from execution. NVIDIA calls this the SIMT model.

The memory hierarchy on a GPU is very simple. Programs have access to an off-GPU global memory that is present on the graphics card. Accesses to this memory are via a high-bandwidth path but suffer from large latencies. Each multiprocessor has a small local store called "shared memory". This space is shared by all the threads running on the multiprocessors. NVIDIA claims that accesses to the local store have latencies comparable to that of accessing registers on the GPU. Finally, each multiprocessor has a very large register file to accommodate the hundreds of threads that may run concurrently.

None of the accesses to the global memory or shared memory go through caches. However, it is possible to designate certain portions of global memory as "textures". These portions can then be accessed through a read-only texture cache.

2) Optimizations: **Local Store**

The GPU does not have a memory hierarchy per se. Accesses to on-device memory are satisfied directly, though with high latencies. Each multiprocessor includes a local store called "shared memory" that is shared by the all its threads and provides low-latency access. Our compiler uses this local store to block/tile memory accesses. The amount of memory we use per thread affects the number of threads that can be run on a single multiprocessor.

GPU Memory Types

Apart from the local store, the GPU contains two other types of fast access structures: a constant cache and a read-only texture cache.

We map read-only input vectors to texture memory when a command-line switch is set. This is because benchmarks that only access their data in a streaming fashion do not show much performance gain from the texture cache. However, some benchmarks gain significantly when their inputs are mapped to textures. Figure 7 shows the speedup achieved when the source vectors for SGEMM and CONV2D are mapped to texture memory. The version of SGEMM that uses the texture cache is $9.3\times$ faster, while CONV2D is $4.9\times$ faster.

Memory Coalescing

Although access to device memory on the GPU is slow, it is via a high-bandwidth path. The GPU also supports coalescing memory requests to improve performance. Coalescing only occurs when certain conditions are met [6, Chapter 5]. Whenever possible, the compiler generates code that meets these conditions.

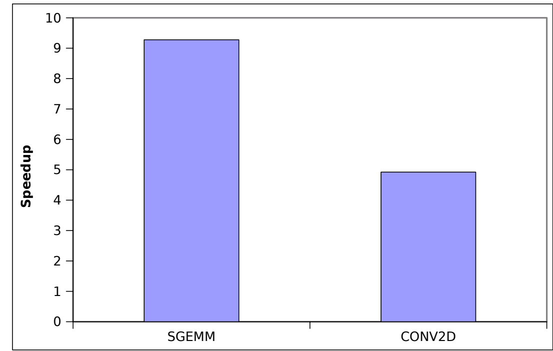

Fig. 7. Accessing input vectors using the texture cache for cache-intensive benchmarks delivers significant gains.

E. Compilation for Heterogeneous Execution

Heterogeneous execution involves running the same DAG across multiple devices concurrently. In many cases, this leads to better utilization of all devices and can be faster than just than execution on a single device.

In this paper, we take a simple approach to generating tasks that can execute concurrently on the CPU and the GPU. DAGs that are part of `forall` loops are natural candidates for heterogeneous execution. These loops are compiled to two threads – one of which executes a fraction of the iterations on the CPU and the other which executes the remainder of the iterations on the GPU. We currently manually use the OpenMP `sections` directive to achieve this as well as the work split.

Other techniques that map generated tasks to multiple devices are available, but are not implemented in the compiler at this point of time. We plan to explore them in the future.

V. Evaluation

A. Benchmarks

We use kernels from the BLAS [21] libraries to evaluate PLASMA against platform-optimized libraries. Both our target platforms – CPU and GPU – have optimized third-party BLAS libraries. The ATLAS 3.6 [22] library is optimized to use the SSE2 instruction set, and also uses parameterized cache blocking. It is not multithreaded. We use NVIDIA's CUBLAS [23] library for our GPU comparisons.

We chose the L1 SAXPY, the L2 SGEMV, and the L3 SGEMM as our test kernels. These kernels are fundamental to BLAS and exhibit behaviour that stresses both the memory and the computation subsystems. These kernels are also building blocks for other BLAS routines. We also evaluate the PLASMA version of the CONV2D kernel.

- SAXPY performs the calculation $Y = \alpha X + \beta Y$. The vectors X and Y contain 1048576 elements each. The SAXPY computation does not reuse any element of X or Y.
- SGEMV performs the matrix–vector calculation $Y = \alpha A \cdot X + \beta Y$. The matrix A contains 1024×1024 elements, and X and Y are of 1024 elements each. SGEMV provides opportunities for "register blocking" [22] as well

Parameter	Value
CPU	2×Intel Quad-core Xeon E5440
RAM	16GB
GPU	NVIDIA 8800 GTS 512
Multiprocessors	16
RAM (per Tesla)	512MB

TABLE II
TEST MACHINE CONFIGURATION

as "cache blocking" [22]. No support for automatic cache blocking is present in our current implementation. As this optimization is closer to the high-level language, we expect these to be implemented at the compiler that transforms the high-level language to our IR. The scalar constants α and β are set to 2.0 and 3.0 respectively

- SGEMM performs the matrix–matrix calculation $C = \alpha AB + \beta C$. All three matrices A, B, and C are of dimensions 1024×1024. The scalar constants α and β are set to 2.0 and 3.0 respectively. SGEMM can also profit significantly from cache blocking.
- CONV2D performs a 2D convolution of a 5×5 filter on a 600×600 image.

Although the number of kernels is low (due to time limitations), other BLAS routines are usually built upon the BLAS kernels evaluated. CONV2D is also a basic operation in many image processing applications.

Each kernel is coded as a single PLASMA source file. Our compiler then translates this source file into multiple C source files for each backend. The gcc 4.3 compiler is then used to compile these C source files with optimization flag "`-O2`". On our 64-bit Linux system, GCC uses the SSE unit for math operations (`-mfpmath=sse`). Our SSE kernels are compiled with the `-msse3` flag, though we use only intrinsics from SSE2. The test machine configuration is detailed in Table V-A

All timings reported in the results include the time taken to transfer data between devices. However time taken to initialize runtime libraries like CUDA is not measured.

B. Comparison to Platform-specific libraries

Figure 8 shows the performance of PLASMA programs compared to platform-specific optimized libraries.

For SAXPY and SGEMV, the PLASMA programs compiled for the CPU (PL-Scalar and PL-SSE3) fare far better than their PL-CUDA counterpart. For SGEMM, however, the PL-CUDA version is $8.6\times$ faster than its CPU counterparts.

As a comparison of PLASMA to platform-optimized libraries, we find that it does not do badly in most cases. The fastest PLASMA SAXPY is almost as fast that of ATLAS, and PL-SSE3 SGEMV is *faster* by about 26%. However, PL-SSE3 SGEMM is $3\times$ slower. ATLAS's SGEMM routines are highly optimized, and in particular have been blocked for optimal cache performance. The kernel output by our compiler is not blocked automatically for cache performance yet.

On the GPU, PL-CUDA SAXPY is about 8% faster than CUBLAS SAXPY. However, PL-CUDA SGEMV on the GPU

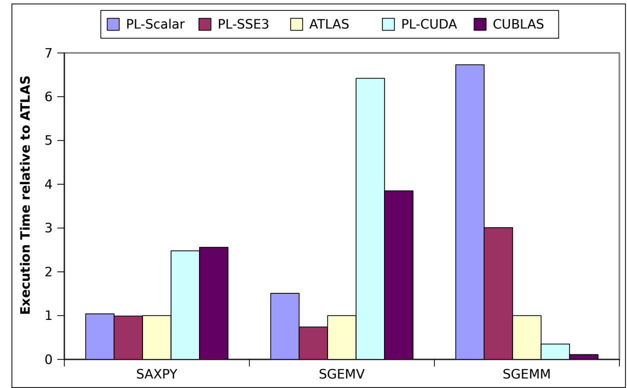

Fig. 8. Performance of PLASMA BLAS versus ATLAS (CPU) and CUBLAS (GPU)

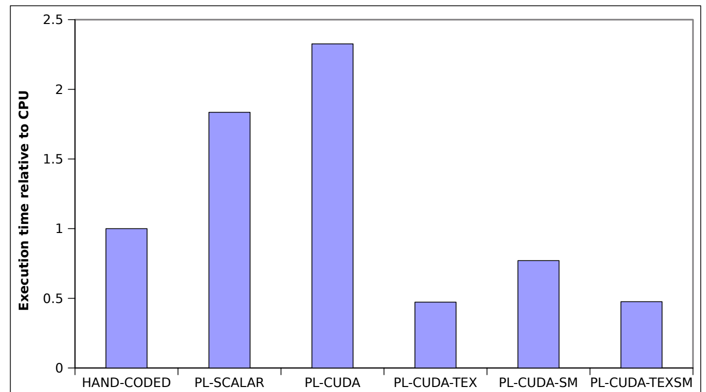

Fig. 9. Performance of hand-coded versus PLASMA CONV2D

is $1.6\times$ slower than its CUBLAS counterpart. Similarly, PL-CUDA SGEMM is $3\times$ slower than CUBLAS SGEMM. In both cases, the hand-optimized CUBLAS functions utilize extensive loop unrolling and use of the local store for blocking. These optimizations are not yet supported by our compiler.

We also hand-code a CONV2D kernel for the CPU and compare its performance (Figure 9) to a PLASMA CONV2D kernel. On the CPU, the hand-coded CONV2D is nearly twice as fast as the PL-SCALAR (plain C) variant. On the GPU, however, the PL-CUDA-TEX version, which uses textures for the source image and filter, is twice as fast as the CPU variant. Unfortunately we do not have a hand-coded GPU CONV2D. The PL-CUDA variant, which does not use textures, is slower than both the hand-coded CPU and PL-SCALAR variants. The PL-CUDA-SM variant has been modified manually (from the CUDA variant) to store and access the filter matrix from the multiprocessors' shared memory. It is nearly as fast as the PL-CUDA-TEX variant, and is faster than the PL-CUDA variant by $3\times$. The PL-CUDA-TEXSM variant accesses the image using the texture cache and the filter using the shared memory, and performs nearly equal to the PL-CUDA-TEX variant. Clearly, for the GPU, memory layout can significantly affect the performance of an algorithm.

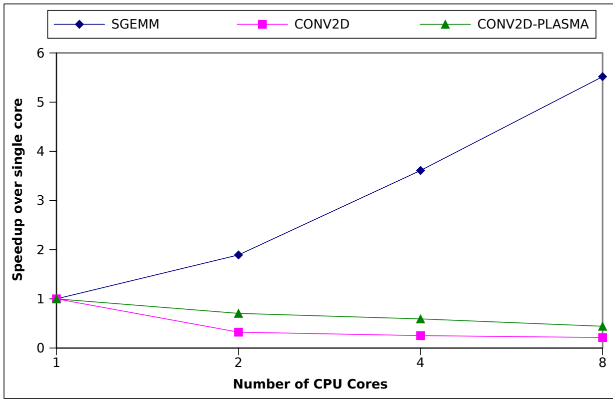

Fig. 10. Speedup for multithreaded PLASMA SGEMM, hand-coded CONV2D and PLASMA CONV2D

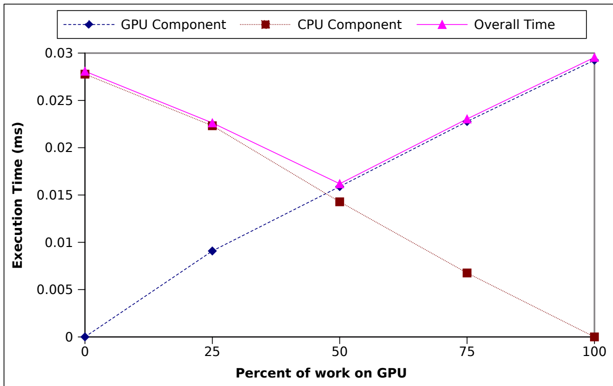

Fig. 11. Execution Time for CONV2D running concurrently across the GPU and CPU

C. Automatic Multithreading

Based on a command-line switch, our compiler can generate OpenMP annotated code for multithreaded execution from the same source code. We currently annotate the outermost `forall` loop with the OpenMP `parallel for schedule(static)` directive. Figure 10 shows the performance of SGEMM and CONV2D running on 1, 2, 4 and 8 CPU cores.

PLASMA SGEMM performance is faster by $5.5\times$ over its single-core variant *without* any changes to the source code.

However, PLASMA CONV2D *slows* down when its outermost loop is distributed across multiple cores. This is true even of the hand-coded CPU variant. It turns out that distributing the outermost loop of CONV2D across cores significantly increases the number of cache misses lowering performance.

D. Heterogeneous Execution

To demonstrate heterogeneous execution we use the PLASMA CONV2D kernel. Specifically, we use the PL-SCALAR and PL-CUDA variants. These variants execute in about the same time (about a 20% difference, see Figure 9).

The main loop of CONV2D is a `forall` loop over the output matrix. We split this loop across the CPU and GPU as detailed in Section IV-B3. We compile these using the techniques of Section IV-E.

Figure 11 shows the results of heterogeneous execution of this kernel. Execution time is the least when the split is 50–50 as we would expect.

VI. Related Work

The VCODE [11] intermediate representation was an effort to compile the many data-parallel high-level languages to a single representation, which would then be compiled to a target architecture. Recently, Vector LLVA [12] and Liquid-SIMD [24] have been proposed to make programs that use multimedia extensions (short vector instructions) in CPUs portable across different architectures. The Stream Virtual Machine [25] was developed as a common target for the many stream programming model based languages. None of these address the issues of distributed memory or that of multiple devices.

Efforts have been made to high-level languages directly to the GPU. The MS Accelerator [16] project provides a library that allows .NET programs to use the GPU without writing GPU shader code. The OpenMP-to-GPGPU [26] compiler compiles an existing OpenMP program to the GPU. StreamIT [8] has been compiled to the GPU [27] with the authors also demonstrating heterogeneous execution of stream programs across the CPU and GPU [28]. The MCUDA [29] project compiles a CUDA program to execute on the CPU.

IBM's Octopiler [4], [30] is a vectorizing compiler that uses a virtual vector representation to compile code to either the PowerPC or the SPE unit of a Cell processor.

VII. Conclusion

Programming accelerators today is harder than it should be. Part of the problem is the enormous diversity in the architecture and programming models of each accelerator. We have demonstrated PLASMA, a framework that enables portable programming across the CPU and GPU. By compiling high-level programs to the IR we have designed, we have shown that it is still possible to get generic code to perform as well as platform-optimized libraries. We have also demonstrated heterogeneous execution – a technique for executing code seamlessly across the CPU and GPU – of PLASMA code. Eventually we hope to compile a number of high-level parallel code to the PLASMA IR thereby enabling existing code to be migrated to accelerators.

Acknowledgements

The authors gratefully acknowledge funding from the NVIDIA Professor Partnership program and Microsoft Research. We also thank members of the Lab for High Performance Computing, SERC, for their feedback on ideas in this paper.

REFERENCES

[1] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the CELL multiprocessor," *IBM J. Res. Dev.*, vol. 49, pp. 589–604, 2005.

[2] ClearSpeed, "ClearSpeed," *Website: http://www.clearspeed.com/*.

[3] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, pp. 39–55, 2008.

[4] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing compiler for the CELL processor," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005, pp. 161–172.

[5] Freescale, *AltiVec Technology Programming Manual*.

[6] NVIDIA, *NVIDIA CUDA: Compute Unified Device Architecture: Programming Guide, Version 2.0*, 2008.

[7] K. Group, "OpenCL," *Website: http://www.khronos.org/opencl/*.

[8] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe, "StreamIt: A compiler for streaming applications," *MIT-LCS Technical Memo TM-622*, vol. 189, p. 195, 2001.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, pp. 777–786, 2004.

[10] M. Baron, "Cortex-A8: High speed, low power," *Microprocessor Report*, vol. 11, pp. 1–6, 2005.

[11] G. Blelloch and S. Chatterjee, "Vcode: A data-parallel intermediate language," in *In Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, 1990, pp. 471–480.

[12] R. L. Bocchino, Jr. and V. S. Adve, "Vector llva: a virtual vector instruction set for media processing," in *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*. New York, NY, USA: ACM, 2006, pp. 46–56.

[13] S. Chatterjee, G. E. Blelloch, and M. Zagha, "Scan primitives for vector computers," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 666–675.

[14] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 97–106.

[15] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, "Overview of the blue gene/l system architecture," *IBM Journal of Research and Development*, vol. 49, no. 2-3, pp. 195–212, 2005.

[16] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program GPUs for general-purpose uses," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006, pp. 325–335.

[17] M. D. McCool, "Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform," *GSPx Multicore Applications Conference*, 2006.

[18] M. D. McCool, Z. Qin, and T. S. Popa, "Shader metaprogramming," in *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002, pp. 57–68.

[19] M. Wolfe, "Vector optimization vs vectorization," *J. Parallel Distrib. Comput.*, vol. 5, no. 5, pp. 551–567, 1988.

[20] NVIDIA, "CUDA: Compute unified device architecture," *Website: http://developer. nvidia. com/cuda*.

[21] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, pp. 308–323, 1979.

[22] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, pp. 3–35, 2001.

[23] NVIDIA, "Cublas library," *Website: http://developer. download. nvidia. com/compute /cuda /1 1/ CUBLAS Library 1.1.pdf*.

[24] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner, "Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, 2007, pp. 216–227.

[25] F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz, "The Stream Virtual Machine," in *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*, 2004, pp. 267–277.

[26] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2009, pp. 101–110.

[27] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software pipelined execution of stream programs on GPUs," in *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 200–209.

[28] ——, "Synergistic execution of stream programs on multicores with accelerators," in *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA: ACM, 2009, pp. 99–108.

[29] J. Stratton, S. Stone, and W. mei Hwu, "Mcuda: An efficient implementation of cuda kernels on multi-cores," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-08-01, March 2008.

[30] IBM, "Compiler technology for scalable architectures," *Website: http://domino.research.ibm.com /comm /research projects.nsf /pages /cellcompiler.index.html*.