Presented at the GSPx Multicore Applications Conference, Santa Clara, October 31 to November 2, 2006.

Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform

Michael D. McCool, RapidMind Inc.
470 Weber St N, Waterloo Ontario, Canada, N2L 6J2
Phone: +1 519 885-5455 Fax: +1 519 885-1463
mmccool@rapidmind.net

Abstract – **The Cell BE processor is capable of achieving very high levels of performance via parallel computation. The processors in video accelerators, known as GPUs, are also high performance parallel processors. The RapidMind Development Platform provides a simple data-parallel model of execution that is easy to understand and learn, is usable from any ISO standard C++ program without any special extensions, maps efficiently onto the capabilities of both the Cell BE processor and GPUs, and can be extended to other multicore processors in the future.**

The RapidMind platform acts as an embedded programming language inside C++. It is built around a small set of types that can be used to capture and specify arbitrary computations. Arbitrary functions, including control flow, can be specified dynamically. Parallel execution is primarily invoked by applying these functions to arrays, generating new arrays. Access patterns on arrays allow data to be collected and redistributed. Collective operations, such as scatter, gather, and programmable reduction, support other standard parallel communication patterns and complete the programming model.

I. INTRODUCTION

The Cell BE processor is an example of a multicore processor, in which several processing elements are integrated onto a single physical chip. It includes nine cores, including a PowerPC core called the PPE and eight vector co-processors called SPEs. The PPE is designed for general purpose computing, including running an operating system, and includes standard features such as a cache. The SPEs, in contrast, are specialized for high performance numerical computation. They do not include a cache, but instead use a local memory and require explicit DMA transfers to and from a shared DRAM. The Cell BE is designed for high-performance parallel computing, and includes features to support several different styles of parallel computing: SIMD within a register (SWAR) and deep pipelining support small scale data parallelism; mailboxes and other synchronization and communication features support task parallelism; and a high-speed interconnect permits multiple Cell BE processors to be connected in order to build larger scale parallel machines.

In addition to the Cell BE, several other multicore processor architectures are emerging. Standard CPUs from IBM, Intel, and AMD are now available in multicore versions. In short order even ordinary desktops may easily have 16 or more processor cores. Video accelerator cards, standard components of most desktop PCs, also support high-performance processors developed by NVIDIA and ATI called GPUs. GPUs are designed for real-time graphics and media manipulation but can also be used for arbitrary computation. High-end GPUs can include up to 48 cores. Both CPUs and GPUs also include SWAR parallelism.

Unfortunately, use of a parallel processor does not automatically lead to higher application performance, especially when a larger number of cores are considered. Existing application code may not expose sufficient parallelism to scale well to larger numbers of cores. Some programming effort will often be required to take advantage of parallel processing.

Parallel programming is different from and frequently more difficult than serial programming. Choices of algorithms, software architectures, languages, development tools, and programming techniques need to be reconsidered. These choices can have an impact not only on the performance of the resulting applications, but also on their robustness and portability. Parallel programming brings a new set of debugging challenges, including the possibility for deadlock, timing, and synchronization errors. Also, the hardware design space for parallel multicore processors is large and diverse, which means that code written at too low a level of abstraction will not be portable to another parallel machine.

Software tools can substantially reduce the effort to parallelize code, and can provide a layer of abstraction that can enhance portability. Automatic parallelization of existing code would be an ideal solution. Unfortunately, automatic parallelization is fundamentally limited by the fact that the best parallel algorithm to solve a problem may be substantially different than the best serial algorithm solving the same problem. It is also hampered by the fact that existing programming languages were not designed to express parallel computations. Parallel execution of loops, for instance, can be prohibited by subtle language features such as pointer aliasing. This fact means that small and seemingly innocuous changes to a program can inhibit automatic parallelization. To express parallel algorithms in a maintainable way, while achieving consistent performance improvements, an explicit approach is desirable. At the same time, deploying completely new programming languages to express parallel computation would involve an unacceptably large retooling, conversion, and training cost. An incremental approach that builds on current practice would be more desirable.

Our goal was to design a programming platform to address these challenges. The RapidMind Development Platform addresses multiple issues from the specification of parallel algorithms via a high-level abstraction, through dynamic code generation for multiple specialized hardware targets, through runtime management of parallel processes. We want to enable developers to quickly build portable, high-performance applications using a simple, safe, and easy-to-use explicitly parallel programming model. In particular, we wanted to give developers access to the power of both the Cell BE and the GPU, but at a high level, so they can focus on developing efficient parallel algorithms instead of managing low-level, architecture-specific details.

The RapidMind platform has several unique features.

First, it has a C++ interface for describing computation, rather than a separate language. The platform interface is implemented using only ISO standard C++ features and works with any ISO standard C++ compiler. It can be used just by including a header file and linking to a library. Although the interface is implemented as a library, the platform enables more than the execution of prebuilt functions. The platform interface permits the expression of *arbitrary* computation. In fact, the system includes a staged, dynamic compiler supporting runtime code generation. This enables some novel high-performance programming techniques above and beyond the exploitation of parallelism. It should also be emphasized that although the interface is implemented in C++, the platform has its own compiler and runtime. The execution model is more like FORTRAN than C++, and this enables the code generated by the system to be aggressively optimized for performance. The unique interface exploits dynamic code generation to permit algorithms to be structured using all the modularization capabilities of C++ *without* incurring a runtime cost for this structure.

Second, the RapidMind platform includes an extensive runtime component as well as interface and dynamic compilation components. The runtime component automates common tasks such as task queuing, data streaming, data transfer, synchronization, and load-balancing. It asynchronously manages tasks executing on remote processors and manages data transfers to and from distributed memory. This runtime component provides a framework for efficient parallel execution of the computation specified by the main program. The platform interface library which is actually linked into the host application is designed to be simple and easy to learn. It mimics a simple matrix-vector library, but this is only a "user interface" to a sophisticated dynamic compiler and runtime system. The interface appears simple only because the runtime is automating the low-level details of parallel execution.

Third, the programming model is safe. Programs written with our platform cannot suffer from deadlock, read-write hazards, or synchronization errors. The platform uses a bulk-synchronization model that supports a conceptual single thread of control, making debugging straightforward. The structure of the language makes parallelism explicit, however, encouraging the development and use of efficient and scalable parallel algorithms.

Fourth, unlike alternatives such as OpenMP, MPI, and threads, our parallel programming model is portable to a wide range of parallel hardware architectures, including vector and stream machines, such as GPUs, as well as distributed memory machines, such as the Cell BE. The system provides a strong execution and data abstraction that is simultaneously modular, portable, and efficient.

In the following, we will first describe the programming model and interface of the RapidMind platform in detail, and then give some examples of its application. We will finally describe, at a high level, how this programming model is mapped onto both the Cell BE processor and GPUs.

II. Related Work

The RapidMind platform interface is an evolution of an earlier system called Sh [4,5,6]. This system established the technology that is the foundation of the RapidMind platform's innovative metaprogramming interface. However, Sh was a research system primarily intended as an advanced shading language for GPUs. In contrast, the commercially developed and supported RapidMind platform is designed to target multiple hardware platforms, and has a more general and complete programming model. In particular, the RapidMind platform makes arrays first-class types, and includes collective operations on arrays.

The RapidMind programming model was inspired by work on abstract parallel computational models as well as previous languages and notations for parallel computation [1,2]. Its computational model has also been based in part on the model of stream programming [7] used on GPUs. It uses an SPMD model, not a pure SIMD model: program objects may include control flow. Its use of an SPMD execution model enables it to emulate certain forms of task parallelism, even though it is primarily a data-parallel model. Functional languages such as SETL and NESL have also influenced the design: RapidMind program objects are pure functions, and advanced programming language features such as partial evaluation are supported. However, the RapidMind platform is not a purely functional programming model, but rather a hybrid supporting both functional and imperative styles of programming.

III. Programming Model and Interface

The RapidMind interface is designed to be as direct an expression of our conceptual data-parallel programming model as possible. It is simplest, therefore, to introduce both at the same time. Our introduction will be relatively detailed; the interface is simple enough that it is possible to convey a working knowledge of it even in the limited space available here.

The interface is based on three main C++ types: **Value<*N*,*T*>**, **Array<*D*,*T*>**, and **Program**. All three types are containers: the first two are for data, the last is for operations. Parallel computations are invoked by applying programs to arrays to create new arrays, or by applying a parallel collective operation that can be parameterized with a program object, such as a reduction.

Upon first glance, the value and array types will not appear to be especially interesting. Nearly every C++ programmer has used an ***N***-tuple type to express numerical computations on vectors, and an array type is also a common way to encapsulate bounds checking or boundary wrapping modes in C++ code. Appearances are deceiving, however. The semantics of these types have been intentionally *chosen* to mimic such common usage in order to make them easy to learn. However, these types together form an interface to a powerful parallel execution engine based on dynamic code generation. This is accomplished by using the **Program** type, which is the key innovation of our interface. A symbolic "retained" mode of execution can be used to dynamically collect arbitrary operations on values and arrays into program objects. Once collected, these operations can be explicitly manipulated by the dynamic compiler in the platform.

VALUES

The **Value<*N*,*T*>** type is an ***N***-tuple. Instances of this type holds ***N*** values of type ***T***, where ***T*** can be a basic numerical type: single-precision floating point, double precision floating point, or signed and unsigned integers of various lengths. Half-precision floating point numbers with 10-bit significands and 5-bit exponents are also supported. Short form declarations are supported for tuples of up to length four using a system of suffixes for the component type. For instance, a 4-tuple of single precision floating point numbers can be specified using **Value4f**; a 3-tuple of unsigned bytes can be specified using **Value3ub**. Booleans may also be used as component types, using the suffix **bool**: a **Value4bool** is a 4-tuple of Booleans.

The standard arithmetic, bitwise, comparison, and logical operators are overloaded on value tuple types and operate componentwise. Standard library functions are also available, such as trigonometric, logarithmic, and exponential functions. These have the same names as the C math library functions whenever possible, but have been extended to also act componentwise on value tuple types. The **Value1f** type can usually be substituted for a **float**; for instance, complex numbers can be implemented on any of the compilation targets supported by the platform using the standard template library: **std::complex<Value1f>**.

In addition to componentwise operations, instances of a value type may be swizzled and writemasked. Swizzling permits the arbitrary reordering and duplication of components. This is expressed with "repeated" array accesses using the "**( )**" operator, with components numbered starting at 0. For instance, if "**a**" is a **Value<4,float>** floating-point 4-tuple representing an RGBA color, then the expression "**a(2,1,0)**" represents a 3-tuple consisting of the first three components of "**a**" in reverse order: a BGR color. If we want to repeat the G component three times instead, followed by the A component, we can use the expression "**a(1,1,1,3)**". If these expressions appear on the left-hand side of an assignment, they are called writemasking. Conceptually, writemasking returns an array of references to components. Computation and assignment on tuples uses parallel semantics: conceptually, a new value is computed first, then all components are assigned simultaneously to the output tuple. Duplication of components in a writemask is illegal since this would indicate that one destination component should be assigned multiple values.

Computations are expressed using value tuples. Operations on value types can be used by the user to directly express SWAR (SIMD within a register) parallelism. Ordinary C++ modularity constructs such as namespaces, classes and functions can be used freely. For instance, the following function computes the reflection of a vector from a 3D plane given by a particular normal:

```
Value3f
reflect (Value3f v, Value3f n) {
  return Value3f(2.0f*dot(n,v)*n - v);
}
```

ARRAYS

The **Array<*D*,*T*>** type is also a data container. However, it is multidimensional and potentially variable in size. The integer ***D*** is the dimensionality and may be 1, 2, or 3; the type ***T*** gives the type of the elements. Currently, element types are restricted to instances of the **Value<*N*,*T*>** type. The number of elements in an array is *not* part of its type.

Instances of the array types support the "**[ ]**" and "**( )**" operators for random access. An element can be retrieved from an array by using a value tuple of the appropriate dimensionality as an argument. The "**[ ]**" operator uses integer-based coordinates, starting at 0. The "**( )**" operator uses real coordinates over the range [0,1] in each dimension, and is useful in conjunction with interpolation modes for tabulated functions and images.

Subarrays can be accessed using **slice**, **offset**, and **stride** functions. Boundary conditions can be set using a **boundary** member function, and include modes to repeat the contents of an array periodically, clamp array accesses to the edge of an array, or return a specified constant for accesses outside the bounds of an array. Interpolation modes can be set similarily.

Arrays, consistent with values, follow by-value semantics. This avoids pointer aliasing and also simplifies programming and optimization. Specifically, arrays can be assigned to other arrays in O(1) constant time, independent of the amount of data in the array or where it is stored. There are two other array-like types, array references and accessors. Array references permit by-reference semantics, which is useful when the destination of a computation is in memory managed by the user. Accessors support references to subarrays. Partial assignment to subarrays referenced through accessors may involve some actual data copying since the contents of the destination array after assignment will be a combination of old data and new data. However, reference counting is used internally to avoid having to do this more than once. Delayed copy-on-write optimizations are also used which enable copying to be eliminated in many circumstances.

Virtual arrays, generated procedurally, are also supported. The **grid** function returns an array of a specified number of elements in each direction whose elements are the same as its indices, but that consumes no storage. Instead, the indices are generated on-the-fly when the array is used as a streaming input. When used for random access, the index value used for the lookup is simply returned directly.

PROGRAMS

The last fundamental type is the **Program**. A program object stores a sequence of operations. These operations are specified by switching to *retained mode*, which is indicated by using the **BEGIN** keyword macro. Normally the system operates in immediate mode. In immediate mode, operations on a value tuple take place when specified as in a normal matrix-vector library: the computation will be performed on the same processor as the host program and the numerical result will be stored in the output value tuple. Entering retained mode creates a new program object, which is returned by the **BEGIN** macro. While in retained mode, operations on value tuples are not executed; they are instead *symbolically evaluated* and *stored* in the program object. Retained mode is exited using the **END** macro, which closes the program object and marks it as being ready for compilation. A completed program object can then be used for computation. Program objects are compiled using a dynamic compiler that can take advantage of the low-level features of the target machine.

It should be emphasized that although the value and array types are wrapped in C++ classes, the execution model of the RapidMind platform is more like FORTRAN, and the compiler used to generate the machine code for program objects can make similar aggressive optimizations. The object-oriented features of C++ can be used to structure computation, but the overhead due to pointer chasing and virtual function calls (for instance) does *not* appear in the final code generated by the platform library. Only operations that actually touch array and value types, and so store symbolic operations in the program object during retained mode execution, actually result in computation on the designated target. The C++ language provides scaffolding for generating the code, but is not itself used for execution.

Here is an example, using the **reflect** function defined earlier as well as the library function **normalize**, which divides a value type by its Euclidean length:

```
Value3f n;
Program p = BEGIN {
    In<Value3f> v;
    Out<Value3f> r;
    Value3f nn = normalize(n);
    r = reflect(v,nn);
} END;
```

This example actually demonstrates a few other concepts. First, the declaration of the "**v**" and "**r**" tuples have been modified by **In<*T*>** and **Out<*T*>** template wrappers. These wrappers mark certain values as the inputs and outputs of the program. Before a program is executed, the values marked as inputs will be initialized with actual data. When a program's execution completes, the final values of the variables marked as outputs are taken as the output of the program.

Note that the non-local variable "**n**" is referenced from inside the program object's definition. When the program object "**p**" is later executed, it will use the current value of "**n**" at the time of execution, whatever it happens to be at that point. This is done even if the program is executing on a remote processor or in parallel on multiple processors. The platform will take care of recording which programs depend on which data and will distribute updated values as needed. It should also be noted that the **normalize** operation here applies directly to a non-local variable. One of the optimizations the platform's compiler can do automatically is lift such "uniform" computations out of parallel computations, so they can be executed once only, and only when the uniform they depend on is itself updated.

A program, once created, is executed by applying it to either tuples or arrays. Let "**V**" and "**R**" be arrays; they can be any dimensionality, as long as it is consistent. The following expression applies the computation specified in the program object "**p**" to all elements of "**V**", creating a new array "**R**" the same size as "**V**". This invokes a million instances of the computation specified by the program object "**p**".

```
Array<D,Value3f> V(1000,1000);
// initialize V with data here
Array<D,Value3f> R;
R = p(V);
// read back the result from R here
```

We do not show how "**V**" is initialized with input data, but it could be the result of a previous computation or loaded from user memory. In general, complex algorithms are implemented using a sequence of such calls, and data is kept in array objects until the computation is complete. It is not necessary to provide a size for "**R**"; it will be computed automatically, based on the size of "**V**". Recall that the assignment semantics of arrays is by-value, and assignment between arrays completely replaces any previous contents of the destination array. The size of an array is part of its value. The execution of "**p**" computes a new value for "**R**" and assigns it to that array along with the new data.

Program objects have intentionally restricted execution semantics. In particular, programs are allowed to read from any value or array declared in immediate mode but are ***not*** allowed to modify the value of any variable not declared inside their definition (or to be more precise, such modifications are not visible outside the program once the program execution is complete). The only outputs programs are permitted are given by values marked explicitly with the **Out<*T*>** template. This restriction means that programs are, from an external point of view, pure functions. The instances of the computation invoked by applying a program to an array can execute in any order, and in particular, can be executed in parallel. Assignment semantics are consistent with this parallel execution model: conceptually, the new values of all elements are computed in parallel and then the new values are assigned simultaneously. The same array can be used as both an input and an output. In this case, the input to the program will always be the "old" contents of the array. The system will automatically double-buffer the output if necessary to avoid read-write hazards.

Programs can also use control flow and can call other programs as subroutines. Since they can include control flow, programs do not execute with a pure SIMD (single instruction multiple data) model, but with the more general SPMD (single program multiple data) model. Recall however that the definition of program objects actually occurs at C++ runtime. Normal C++ control flow can manipulate the generation of code, since it affects what operations are recorded, but does not result in actual control flow when the program executes. To support dynamic control flow, which can depend on values actually computed by the program, macros are supplied that can insert appropriate control-flow code into the program object. These macros simply wrap some function calls into the platform API but make the syntax cleaner. The example below is a simple particle system that moves particles along a ballistic path until their maximum lifetime is achieved, then resets them with a position at zero and a new velocity that is a hash of their final position:

```
Value3f acc;            // acceleration
Value1f del_time;       // time increment
Value1f max_time;       // maximum particle lifetime
Program evolve = BEGIN {
    In<Value3f> vel,  pos;
    In<Value1f> time;
    Out<Value3f> new_vel,  new_pos;
    Out<Value1f> new_time;
    new_vel = vel + acc * del_time;   // update velocity
    new_pos = pos + vel * del_time;   // update position
    new_time = time + del_time;       // update time
    IF (time > max_time) {            // reset if expired
      new_pos = Value3f(0,0,0);
      new_vel = normalize(hash(pos));
    } ENDIF;
} END;
```

In addition to **IF**/**ELSEIF**/**ELSE**/**ENDIF**, the system also supports loops with **FOR**/**ENDFOR**, **WHILE**/**ENDWHILE**, and **DO**/**UNTIL**. The control arguments to these keywords can themselves include arbitrary code including control flow (which can occur if the control expression in an argument calls a function).

This program can be expressed in a slightly more concise way as follows, where **InOut<*T*>** simply marks a certain type as *both* an input and an output:

```
Value3f acc;            // global acceleration
Value1f del_time;       // time increment
Value1f max_time;       // maximum particle lifetime
Program evolve = BEGIN {
    InOut<Value3f> vel,  pos;
    InOut<Value1f> time;
    vel += acc * del_time;    // update velocity
    pos += vel * del_time;    // update position
    time += del_time;         // update time
    IF (time > max_time) {    // reset if expired
      pos = Value3f(0,0,0);
      vel = normalize(hash(pos));
    } ENDIF;
} END;
```

The **InOut<*T*>** marker used here does *not*, however, mean that the actual input passed to the program object will be modified. Inputs are always passed by value, never by reference. The input and output "slots" will be separate.

This program also returns multiple values. Actually, programs always return an instance of the **Bundle** type which is a container for one or more arrays or tuples. A bundle with only one array (or tuple) component can be assigned to an array (or tuple) with automatic conversion, but programs that return multiple values must be assigned to bundles to distribute their return values to multiple target arrays. The **bundle** function makes it easy to create temporary bundles as needed. Instances of the **Bundle** type rarely have to be declared explicitly. This is demonstrated in the following, which computes one update of the state of the particle system:

```
Array<1,Value3f> Vel, Pos;
Array<1,Value1f> Time;
bundle(Vel, Pos, Time) = evolve(Vel, Pos, Time);
```

This example also demonstrates the parallel assignment semantics. The output arrays are the same as the input arrays, but the new values are not assigned until all the old values have been consumed.

Partial Evaluation and the Program Algebra

Program objects are containers for operations, and these operations can be manipulated by the system explicitly. This permits the implementation of several advanced programming features.

First, programs can be partially evaluated. Given a program object "**p**" with *n* inputs, the expression "**p(A)**" returns a program object with *n*-1 inputs. In other words, the inputs to program objects do not need to all be provided at once. It is also possible to bind all the inputs to a program object, but defer its actual execution. Execution of a program object is only triggered when it is assigned to a bundle, an array, or a value.

So far we have used the "**( )**" operator to bind inputs to program objects. This is called *tight binding*. Changes to the input after tight binding will not affect the input to the program, even if actual execution is deferred. In the above example, suppose we create "**p(A)**" and assign it to a program object "**q**", then modify the input "**A**". When we later execute "**q**", it will still use the value of "**A**" that was in effect when it was bound to "**p**", not the new value. Tight binding enables optimization of the program object based on the actual values in the bound input.

However, the system also supports *loose binding*, specified with the operator "<<". The expression "**p << A**" is similar to that of "**p(A)**", except that changes to "**A**" *will* be visible to a deferred execution of "**p << A**".

An input to a program can be bound to either an array or a value tuple. If arrays of different sizes are bound to a program, the smaller inputs are replicated according to their boundary conditions to match the size of the largest input. If both an array and a value tuple are bound, the tuple is duplicated to match every element of the array. Let "**p**" have two inputs and suppose we execute "**q = p << a**" where "**a**" is a value tuple. If "**q**" is latter applied to an array "**B**" and assigned to an output array "**C**", for example using "**C = q(B)**", then the value "**a**" in affect at the time will be uniformly duplicated to match all elements of "**B**". In other words, "**a**" will have precisely the same semantics as if it had been referenced as a non-local variable. Loose binding is invertible: the operator ">>" can undo a loose binding. Suppose "**p**" refers to a non-local variable "**a**". The expression "**q = p >> a**" creates a new program object "**q**" with an extra input. This extra input is used internally in place of all previous references to "**a**". These operators are very useful for managing the interface of program objects.

Second, programs can be combined to create new programs using two operations: functional composition and bundling. These operations form a closed algebra over program objects [6]. The functional composition operator "<<", when applied to two program objects "**p**" and "**q**" as in "**p << q**", feeds all the outputs of the program on the right to the inputs of the program on the left. It creates a new program object with the inputs of "**q**" and the outputs of "**p**". The bundle operation uses the **bundle** function. It concatenates all the inputs and outputs of its arguments in order. It creates a new program that is equivalent to concatenating the source of its input programs in sequence. Together, and when combined with the dynamic compiler, these two operations are very powerful. For instance, a simple program can be defined that discards some of the outputs of another program, and this can be used, in conjunction with the dead-code eliminator in the dynamic compiler, to implement *program specialization*. The inputs to a program can also be reordered using another simple program in combination with the "<<" operator. This *interface adaptation* will have no runtime cost in the final compiled program. Program algebra operations can also be used to combine stages of a pipeline into a single, more efficient pass.

Collective Operations

The parallel application of a program object to an array is the basic form of parallel computation supported. However, other patterns of communication and computation are available in the form of collective operations. Completely general patterns of communication are supported with the scatter and gather operations, and the reduction operation provides a hierarchical computational pattern. Application and reduction are compared in Fig 1. The combination of collective operations with the general parallel apply operation is a general programming model that can be used for the efficient implementation of a wide range of algorithms [1,2].

A gather operation reads from random locations in an array. It takes both a data array and an index array as input, and generates an output array. It is a parallel operation, and can be invoked with the following syntax:

```
A = B[U];
```

where all of **A**, **B**, and **U** are arrays. Note that **A** will have the size of **U** and must also have its dimensionality. The element type of **A** must also match the element type of **B**. Since random access into arrays is also possible from inside programs, the gather operation can be emulated with appropriate program object definitions. The inclusion of a separate gather operator in the language is only for convenience and for symmetry with scatter.

The scatter operation is similar to gather, but the destination rather than the source addresses are given, and the "**[ ]**" operator appears on the left:

```
A[U] = B;
```

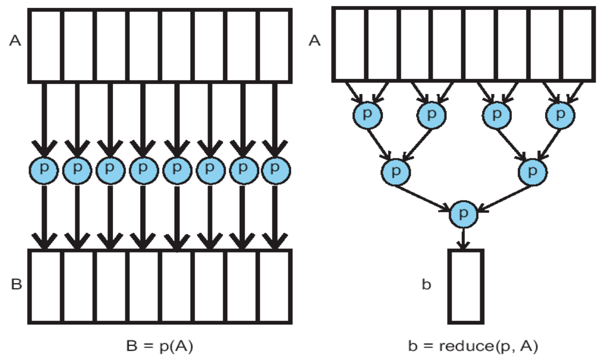

Fig. 1: Communication patterns for application and reduction.

Since program objects do not support random access write, the scatter operation cannot be emulated with a simple program. With some difficulty scatter can be emulated using sorting and search to convert it into a gather. However, including scatter in the language permits a more efficient, platform-specific implementation.

For non-deterministic behavior, the elements of **U** in a scatter should be distinct. Two properties are guaranteed: first, any element of **A** whose address is not specified in **U** will not be modified; second, if the same element of **A** is a destination for multiple elements of **B**, that is, if some elements of **U** are non-unique, the writes will be atomic, and precisely one of the elements from the set written to a common destination will survive. For deterministic computation, this latter feature should be avoided and unique destination addresses should be used.

A programmable reduction function is also provided. A reduction combines elements of an array using an associative operator with two inputs and one output, all of the same type. For example, a reduction based on the addition operator would sum all the elements of an array, returning a single element. A reduction can be implemented in O(log *n*) time using only the features described so far, so including it in the interface is partly a convenience feature. However, including it as an intrinsic operation raises the level of abstraction and may permit use of a more efficient implementation.

The most general form of the **reduce** function takes a program argument. This program should have two inputs and one output, all of the same type. The reduction returns a single instance of the element type of the array which is the result of the reduction. For example, if "**$**" is an associative operator on type ***T***, the following performs a reduction using that operator:

```
Program combiner = BEGIN {
  In<T> a, b;
  Out<T> r = a $ b;
} END;
Array<D,T> A;
T result = reduce(combiner,A);
```

The combiner function can include arbitrary code, but the system makes no attempt to determine if the combiner operation is associative. If it is not, different hardware platforms may produce different results, and in fact, different results may be produced on different runs even on the same hardware platform. For deterministic results, an associative operator must be used. Certain reductions are common and these are built in: **sum**, **product**, **min**, and **max** are supported. Although not purely speaking reductions, **arg_min** and **arg_max** are also supported. They return the index of the corresponding extreme as well as its value.

Additional collective operations are supported; for instance, it is possible to count the number of elements that satisfy a certain predicate in an array. There are also various forms of scatter (with different rules for resolving collisions) and reduction (for instance, reduction of only the columns of an array).

IV. Usage

To clarify the usage of our system, we will now give an example that shows how a simple loop nest may be parallelized. Suppose we have been given the following simple application code:

```
float f;
float a[512][512][3];
float b[512][512][3];

float func (float r, float s) {
   return (r + s) * f;
}

void func_arrays ( ) {
  for (int x = 0; x<512; x++)
    for (int y = 0; y<512; y++) {
      for (int k = 0; k<3; k++) {
        a[y][x][k] = func(a[y][x][k],b[y][x][k]);
      }
    }
  }
}
```

This computation can be expressed using the RapidMind platform interface as follows:

```
#include <rapidmind/platform.hpp>

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);

void func_arrays ( ) {
  Program func_prog = BEGIN {       // define program
    In<Value3f> r, s;
    Out<Value3f> q;
    q = (r + s) * f;
 } END;
 a = func_prog(a,b);               // execute program
}
```

Program objects are defined and compiled dynamically. While not especially slow, program definition and compilation should not be placed in an inner loop or repeated unnecessarily. The above can therefore be made more efficient by marking the program object static and including a definition guard to ensure the program is only constructed the first time the "**func_arrays**" function is called. An alternative structure to avoid repeated dynamic definitions of program objects, which we call a *processor pattern*, is to include the definition of program objects in the constructor of a class, and then use static instances of this class. The compiled program objects can be stored as member variables in the object, then a member function can be used to apply these programs to data. This is also a good pattern to use when generating parameterized code.

Space prevents us from giving more detailed examples. Unfortunately, the simple examples we have given above can be misleading, as they do not show the full power of the system for structuring large applications. It is important to realize that the definition of programs is given by the *dynamic* sequence of operations executed by the host while in retained mode. Which operations get captured and stored in program objects is given by the flow of control in the host program, *not* by syntactic enclosure. Since the platform interface only "sees" operations that touch RapidMind types, arbitrary code can be executed by the host program to decide which operations can be included in the program object without slowing down the final execution of the program object. For example, the host program can walk a complex data structure, perhaps a graph or tree representing a desired computation, and issue operations only occasionally, but the platform interface will be able to pack the issued operations together into a compact executable program. Using the platform interface and this concept, it is very easy to turn interpreters into compilers.

V. Applications

Several applications have been implemented on the RapidMind platform, including both simple test cases and complex commercial applications. Commercial applications include a real-time raytracer intended for the visualization of complex real-world CAD data. In this section, we will describe some of these at a high level; please go to **`http://www.rapidmind.net`** for more information.

To test performance, simple benchmark applications have been implemented. These include dense matrix-matrix multiplication, the FFT, and an implementation of Black-Scholes quasi-Monte Carlo option pricing, as well as other tests. The descriptions and performance results for these benchmarks will be made available separately.

Fig. 2 is a screenshot of a real time flocking simulation of a crowd of 16,000 autonomous characters [8]. They can move around a height field and modify their motion based on their environment, their internal state, and their proximity to each other. The actual simulation took place on an IBM Cell BE Blade. The state of the characters was streamed over a network connection to a visualization server, which animated and rendered the characters on a GPU. The RapidMind platform was used to specify and execute *both* the simulation computation and the GPU shaders used for the visualization. In the RapidMind system, GPU shaders are just special cases of program objects.

Flocking requires global communication; any character can potentially interact with any other. The RapidMind platform abstracts the memory system of the Cell BE processor so that simple global scatter and gather memory operations can be used for this communication. The platform provides a sufficient level of abstraction that the simulation prototype was actually prototyped on a GPU by the author in a single afternoon, then ported to the Cell BE the next working day. It is also possible to run the same simulation on a GPU without any changes to the algorithm specification.

Fig. 3 shows an image rendered with a real-time raytracer implemented on top of the RapidMind platform. This raytracer is part of a commercial product developed by RTT AG, and runs in real time on both a GPU and on the Cell BE. Several other advanced graphics applications have been implemented using the platform, including a deferred shading system, a soft shadow algorithm, a technique for representing vector images in texture maps [9], and others.

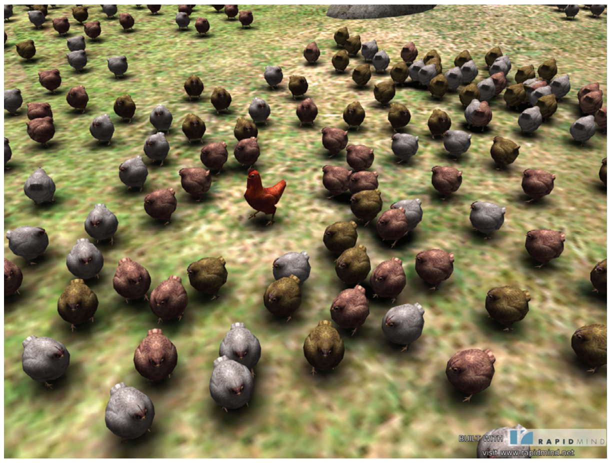

Fig. 2 Real time flocking simulation.

Fig. 3 Image generated by a real-time raytracer implemented on the RapidMind platform.

A vision application that attempts to find the locations of circles of a certain size in an input image was also implemented as an image processing test case. This code has been applied to the problem of tracking microscopic beads in confocal microscopy images obtained as part of a physics experiment [3]; input data is shown in Fig. 5. The tracking algorithm involves an edge detector, a convolution mask over the result of the edge detector, and the use of collective operations to locate the peak responses of the convolution. An interesting feature of this example is the convolution with a circular mask, an example of which is shown in Fig. 4. This convolution mask is not separable, but the mask only has O(n) non-zero values, so it is possible to implement it in O(n) time rather than the O(n^2) time of a naïve implementation. The convolution function in this example exploits the dynamic code generation ability of the system to only tap the input image at the non-zero locations. This optimization, which involves a simple "**if**" statement in the convolution function, works for any sparse convolution mask. The data defining the convolution mask could easily be changed to search for X's, for instance, and the system would generate the appropriate code automatically.

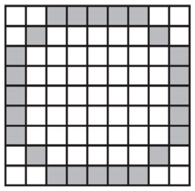

Fig. 4: A sparse convolution mask.

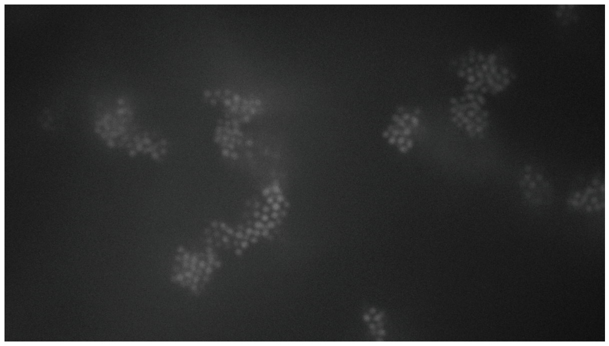

Fig. 5: Confocal microscopy images used as input to the vision application test case. Image courtesy of Peter J. Lu.

VI. Implementation on the Cell BE

Three types of explicit parallelism can be defined by the interface we have described: SWAR parallelism represented by vectorized operations on tuples, elementwise application of programs to arrays, and the alternative forms of parallelism supported by collective operations.

The SWAR parallelism can be mapped directly onto the SIMD registers supported in the Cell BE architecture. It should be noted, however, that not all swizzle operations can be implemented equally efficiently. Also, there are alternative strategies for vectorization, such as combining multiple invocations of the program over multiple elements of arrays, that may ignore the vectorization specified by the user.

Secondly, there is the parallelization specified by applying the program to an array. This can conceptually invoke thousands to millions of potentially parallel computations. Clearly the Cell BE does not have this many cores, so we do not map these computations one-to-one to SPEs. Instead, the work is divided into work packets that access coherent regions of each array. The work packets and their input are then streamed into the SPE processing elements. Triple buffering is used to hide the latency of stream access, and computation takes place in parallel with data transfer. For random access from program objects, a software cache is used. Also, if the program objects include dynamic control flow, different work packets can take different amounts of time to complete, and so dynamic load balancing is used.

Collective operations are supported by specialized compilation and parallel execution strategies, even if they could be implemented using other features of the interface. An example is reduction. Reduction is not really a theoretically “primitive” operation, since it can be specified using other features of the interface to run in O(log *n*) time. However, since it is used so frequently, the platform implements it in a special way on the Cell BE that takes advantage of some of some of the special features of this platform.

VII. Implementation on the GPU

The GPU implementation is based on top of the OpenGL API, and works on both NVIDIA and ATI GPUs. Data is stored in textures and render-to-texture and multiple render target extensions are automatically used when the output of a program is assigned to an array. Several optimizations specific to the OpenGL API are used to obtain good performance. For instance, it is relatively expensive to allocate and deallocate textures under OpenGL, so the system automatically reuses previously allocated textures whenever possible.

The memory model of GPUs is different from that of the Cell BE. In particular, the video memory is most efficiently accessed by DMA transfers to and from the host memory rather than as shared memory. The use of the array type to represent memory permits the system to abstract the actual storage location of data. The system automatically tracks where data was last updated and avoids transfers whenever possible. It is, however, possible to explicitly force a transfer to or from the host memory. The array abstraction and transfer mechanism also makes it possible to extend the system to more general distributed memory systems in the future.

Implementations of benchmarks using the RapidMind platform have shown performance equivalent to the best available GPU implementations done directly through OpenGL. However, more direct access to the hardware would provide opportunities for even higher performance. Recently ATI has announced support for a lower level interface specific to their hardware. This interface exposes certain non-standard features of their hardware that are not available through any existing graphics API. The most important of these features involve memory access and management which could provide opportunities for performance improvements in the future.

Unlike other alternatives for general purpose programming on GPUs, the RapidMind platform may also be used to specify shaders. Shaders are treated as a special case of program objects. In order to integrate with graphics applications, for example to visualize the results of computations, is it is possible to relate arrays to particular textures in the OpenGL interface. However, it is definitely not necessary to use the OpenGL interface in the course of ordinary computation.

VIII. Conclusion

The RapidMind Development Platform combines an interface based on dynamic compilation with a data-parallel programming model. It can be considered either an API for specifying arbitrary parallel computation or a parallel programming language embedded in C++. It supports the explicit specification of various forms of parallelization, including SIMD within a register (SWAR) and parallel execution of SPMD programs. The RapidMind platform uses a portable programming model that maps onto both the Cell BE and GPUs. This programming model has the advantage that no synchronization primitives are required, but it is not possible to write code that includes deadlock or timing errors. This last property, combined with its simple, easy-to-learn semantics, enables access to the performance of the Cell BE and GPU processors for a broad range of programmers.

References

[1] Guy E. Blelloch, 1990. *Vector Models for Data-Parallel Computing.* MIT Press.

[2] Alan Gibbons and Wojciech Rytter, 1988, *Efficient Parallel Algorithms*, Cambridge University Press.

[3] Peter J. Lu, Jacinta C. Conrad, Hans M. Wyss, Andrew B. Schofield, and David A. Weitz, *Fluids of Clusters in Attractive Colloids*, Physical Review Letters, 96/028306, 2006.

[4] Michael D. McCool and Stefanus Du Toit, 2004. *Metaprogramming GPUs with Sh*. AK Peters.

[5] Michael D. McCool, Zheng Qin, and Tiberiu Popa, 2002. *Shader Metaprogramming*. Graphics Hardware, 57–68.

[6] Michael D. McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule, 2004. *Shader Algebra*. SIGGRAPH, 787–795.

[7] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn and Timothy J. Purcell, 2005. *A Survey of General-Purpose Computation on Graphics Hardware.* Eurographics 2005, State of Art Report.

[8] Craig W. Reynolds, 1987. *Flocks, Herds, and Schools: A Distributed Behavioral Model*, SIGGRAPH, 25-34.

[9] Zheng Qin, Michael D. McCool, and Craig Kaplan, 2006. *Real-Time Texture-Mapped Vector Glyphs*. Symposium on Interactive 3D Graphics and Games, 125–132.