Writing Applications for the GPU Using the RapidMind™ Development Platform

Contents

Abstract

The RapidMind Development Platform allows developers to use standard C++ programming to easily create applications targeted for high-performance processors, including GPUs, the Cell BE, and multi-core CPUs. In the case of the GPU, the RapidMind platform can be used for both shaders and general purpose processing. For shaders, the platform provides many advantages over other shading systems, including support for object-oriented programming. For general purpose processing, the platform provides a simple computational model that can be mapped onto any available computational resource in a system, including both GPUs and other processors. Code can be written once, then run in parallel on any of the processors that RapidMind supports.

Introduction

Graphics Processing Units (GPUs), the main processors of commodity video accelerator cards in PCs, are capable of achieving very high levels of performance by utilizing the power of parallel processing. In fact, these processors typically have more than an order of magnitude more floating-point power than the host CPU they support. Although designed for graphics applications, they can be used for arbitrary computations; however, graphics APIs do not directly support this mode of usage.

The RapidMind Development Platform makes it straightforward to access a GPU's power for general-purpose computation without any need to work through a graphics API. The RapidMind platform provides a simple data-parallel model of execution that is easy to understand and use, and maps efficiently onto the capabilities of GPUs. The RapidMind platform's unique interface enables access to a GPU's power from within a single ISO-standard C++ program without any special extensions. It allows the use of familiar development environments and tools by building upon concepts and strategies already familiar to C++ programmers.

It is also possible to use the platform to write shaders and graphical applications as a special case of the platform's general-purpose capabilities. The RapidMind platform supports both NVIDIA® and ATI® GPUs.

This document outlines the challenges general-purpose GPU programming presents to programmers and describes how these challenges are effectively overcome using the RapidMind Development Platform. A simple example of a particle system simulation is used to show how straightforward it is to leverage GPUs for general purpose computation by using the RapidMind Development Platform. We also demonstrate how the platform can be used to program shaders for graphical applications.

Graphics Processing Units

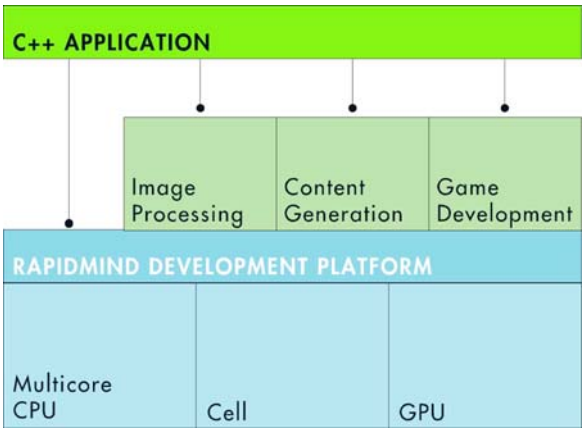
GPUs from ATI and NVIDIA are massively parallel processors that can support more than 30x the floating-point power of typical host CPUs and more than 5x the memory bandwidth. This processing power is necessary to implement shaders, which are small programs which must be executed at every vertex to transform it in 3D space and at every pixel to compute the color of that pixel. Although they are powerful, GPUs are widely available on inexpensive video accelerator cards, and are standard components of most PCs.

The execution of shaders can be considered a form of stream processing, a massively parallel computing model that emphasizes coherent access to memory. However, the programming model for GPUs is usually expressed in graphics terms, and can be inconvenient to use for general-purpose computation. For instance, to apply a function to an array, it is necessary to bind a shader and draw a rectangle to the screen. To implement random-access reads from memory, it is necessary to set up texture maps and bind them to shaders. To write to output arrays, it is necessary to use frame buffer object interfaces to bind the output of the shaders to a texture. To write to *computed* locations in output arrays, it is necessary to reinterpret image pixel values as vertex positions and render a sequence of points that the rasterizer scatters to new locations on a destination buffer. This makes it very complex and confusing to program GPUs for general applications.

Although GPU hardware vendors provide languages for programming the shader units of their hardware, these languages only provide the ability to program shaders—not the whole system. Many other aspects of the GPU must be managed, in particular memory. The GPU memory is separate from the host memory, so it is necessary to transfer data to and from the video accelerator. Also, binding shader code to the host application requires a large amount of glue code. Finally, after programming a GPU this way, the resulting code will only run on GPUs, and cannot easily be ported to other hardware targets, such as the Cell BE.

RapidMind Development Platform

RapidMind provides a software development platform that allows the developer to use standard C++ programming to easily create high-performance and massively parallel applications that run on the GPU. Developers are provided a single, simple and standard way to program which the RapidMind platform then maps onto all available computational resources in a given system. Developers can continue to use their existing C++ compilers and build systems. The RapidMind platform is embedded in the application and transparently manages massively parallel computations.

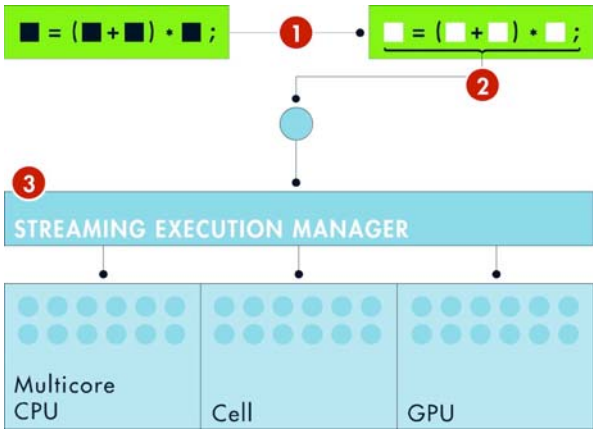


Application Structure

Writing RapidMind Enabled Applications

Users of the RapidMind Development Platform continue to program in C++ using their existing compiler. After identifying components of their application to accelerate, the overall process of integration is as follows:

1. **Replace types**: The developer replaces numerical types representing floating point numbers and integers with the equivalent RapidMind platform types.
2. **Capture computations**: While the user's application is running, sequences of numerical operations invoked by the user's application are captured, recorded, and dynamically compiled to a program object by the RapidMind platform.
3. **Stream execution**: The RapidMind platform runtime is used to manage parallel execution of program objects on the target hardware (in this case, a GPU).



Process to change applications to use the RapidMind Development Platform

How Does It Work?

The RapidMind Development Platform is an advanced dynamic compiler and runtime management system for parallel processing. It has a sophisticated interface embedded within standard ISO C++. It can be used to express arbitrary parallel computations, but it is *not* a new language. Instead, it merely adds a new vocabulary to standard ISO C++: a set of nouns (types) and verbs (operations). In the case of the GPU, this new vocabulary allows the specification of high-performance parallel computations, while retaining close integration with standard C++ code running on the host processor. In essence, the RapidMind Development Platform enables programming of the GPU as a co-processor, under a single unified programming model, while using existing tools.

A user of the RapidMind Development Platform writes C++ code in the usual way, but uses specific types for numbers, vectors of numbers, matrices, and arrays. In immediate mode, operations on these values can be executed on the host processor, in the manner of a simple operator-overloaded matrix-vector library. In this mode, the RapidMind Development Platform simply reflects standard practice in numerical programming under C++.

However, the RapidMind platform also supports a unique retained mode. In this mode, operations are recorded and dynamically compiled into a "program object" rather than being immediately executed. These program objects can be used as functions in the host program. In the case of GPUs, applying such a function to an array of values automatically invokes a massively parallel computation on the video accelerator. Data is automatically transferred to and from the video accelerator, overlapping computation with data transfers. Program objects mimic the behavior of native C++ functions, including support for modularity and scope, so standard C++ object-oriented programming techniques can be leveraged. It should be noted that at runtime, program objects *only* execute the numerical computations they have recorded, *and can completely avoid any overhead due to the object-oriented nature of the specification.* The platform uses C++ only as scaffolding to define computations, but rips away this scaffolding for more efficient runtime execution.

The RapidMind Development Platform can target other processors. In particular, it can also be used to program the Cell BE processor and the host CPU with dynamically compiled code.

Example: A Particle System

As a simple example to demonstrate the use of the RapidMind Development Platform on the GPU, the code below is provided. This code simulates a particle system, and specifies a parallel computation which will take place on the GPU. Every particle has a state. At every timestep of the simulation, each particle must execute a rule to update its own state and such rules can depend on a number of other inputs. Such simulations have many applications, especially when a sufficiently large number of particles are used, as is possible on the GPU. Appropriate choices of state and update rules can give rise to a wide variety of interesting phenomena.

We can store the state of all the particles in an array and then write a "program object" that will update all the states in parallel. A state update rule that integrates the acceleration due to gravity and checks for collision against a ground plane is given in the listing on the next page. More complex rules are possible that check for collision against other objects or that use the state of other nearby particles to implement collective behaviors.

After the simulation step, another rule can be run to convert the state of each particle to a visualization. These rules can also be written using the RapidMind Development Platform.

In this code example, the words in boldface are types and keywords (macros) provided by the RapidMind Development Platform. These types and macros are implemented using standard ISO C++ features, and work with a variety of compilers. The types used in this example come in a number of flavors representing different geometric objects, but basically represent short vectors of floating point numbers. The user can create types for their own purposes by extending the types provided. The **`BEGIN`** macro enters retained mode and creates a program object called `particle`, and marks it for the streaming execution mode. This program

object will act as a container for all the operations specified up to the invocation of the **END** macro. The **InOut** type modifier marks the `pos` and `vel` variables as both inputs and outputs. Finally, at the end of the code example, we invoke the program object `particle` on arrays of positions `ps` and velocities `vs`, updating them. The syntax shown can combine multiple arrays into a single stream, execute the program object, then split apart the output into multiple destination arrays. The computation specified in this code example executes completely on the GPU.

```
// STATE ARRAYS
Array<1,Value3f> ps(n_particles);       // array holding particle positions
Array<1,Value3f> vs(n_particles);       // array holding particle velocities

// PARAMETERS
float mu = 0.3f;              // parameters controlling collision response
float eps = 0.6f;
Value1f delta(1.0f/60.0f);    // timestep delta (this type mimics a float)
Value3f g(0.0f,-9.8f,0.0f);   // acceleration due to gravity

// DEFINE STREAM PROGRAM
particle = BEGIN {   // create a new program object
  InOut<Value3f> pos;   // position; both input and output
  InOut<Value3f> vel;   // velocity; both input and output
 // SPECIFY COMPUTATIONS
  // clamp acceleration to zero if particles at or below ground plane
  Value3f acc = cond(abs(pos[1])<0.05f, Value3f(0.0f,0.0f,0.0f), g);
  // integrate acceleration to get velocity
  vel += acc*delta;
  // integrate velocity to update position
  pos += vel*delta;
  // check if below ground level
  Value1f under = pos[1] < 0.0f;
  // clamp to ground level if necessary
  pos = cond(under, pos * Value3f(1.0f,0.0f,1.0f), pos);
  // modify velocity in case of collision
  Value3f veln = vel * Value3f(0.0f,1.0f,0.0f);
  Value3f velt = vel - veln;
  vel = cond(under, (1.0f - mu)*velt - eps*veln, vel);
  // clamp position to the edge of the plane, just to make it look nice
  pos =
    min(max(pos, Value3f(-5.0f, numeric_limits<float>::min(), -5.0f)),
        Value3f(5.0f, numeric_limits<float>::max(), 5.0f));
} END;

// EXECUTE STREAM PROGRAM WITH MULTIPLE VALUE RETURN BUNDLE
bundle(ps, vs) = particle(ps, vs);   // uses parallel assignment semantics
```

Stream program for particle system simulation

One of the important features of the RapidMind Development Platform is that no glue code is required. The above code *is* the API. Co-processor code can be treated as simply part of the host application. Program objects can be used, essentially, as dynamically definable functions that can be run in parallel on all the elements of arrays. On the GPU, execution of program objects, such as `particle` in the example, are automatically mapped to the graphics hardware and managed by a runtime scheduler.

You will notice that there is no explicit reference in the above program to a graphics API or the GPU. Neither the developer nor the application needs to be concerned about the actual processor resources that might be available at runtime. Whether targeting for a multi-core CPU, a GPU, or the Cell BE, the program itself does not change. The RapidMind Development Platform takes care of mapping the program objects onto available computational resources.

Example: Blinn-Phong Shading

The RapidMind platform can also be used to implement shaders. The listing given below presents an example where the platform is used to specify vertex and fragment shaders in order to implement the Blinn-Phong lighting model. Shaders are GPU-specific, but conceptually they are special versions of the program objects used for general-purpose computation. Functions and objects defined for general-purpose computation can also be used for shaders, and vice-versa. The platform also manages texture maps and other memory, enabling a powerful form of data abstraction not available in other shading languages. Using the platform to program shaders is also useful when the results of general-purpose computations on the GPU need to be visualized.

```
// DEFINE VERTEX SHADER
vertex_shader = BEGIN_PROGRAM("vertex") {
  // declare input vertex attributes (unpacked in order given)
  In<Value3f> nm;    // normal vector (MCS)
  In<Value4f> pm;    // position (MCS)
  // declare outputs vertex attributes (packed in order given)
  Out<Value3f> nv;   // normal (VCS)
  Out<Value3f> pv;   // position (VCS)
  Out<Value3f> pd;   // position (DCS)
  // specify computations (transform positions and normals)
  pv = (MV*pm)(0,1,2);    // VCS position
  pd = (MD*pm)(0,1,2);    // DCS position
  nv = normalize(nm*inverse_MV(0,1,2)(0,1,2)); // VCS normal
} END;

// DEFINE FRAGMENT SHADER
fragment_shader = BEGIN_PROGRAM("fragment") {
  // declare input fragment attributes (unpacked in order given)
  In<Value3f> nv;     // normal (VCS)
  In<Value3f> pv;     // position (VCS)
  In<Value3f> pd;     // fragment position (DCS)
  // declare output fragment attributes (packed in order given)
  Out<Value3f> fc;    // fragment color
  // compute unit normal and view vector
  nv = normalize(nv);
  Value3f vv = -normalize(pv);
  // process each light source
  for (int i = 0; i < NLIGHTS; i++) {
    // compute per-light normalized vectors
    Value3f lv = normalize(light_position[i] - pv);
    Value3f hv = normalize(lv + vv);
    Value3f ec = light_color[i] * max(0.0f,dot(nv,lv));
    // sum up contribution of each light source
    fc += ec *(kd + ks*pow(pos(dot(hv,nv)),q));
  }
} END;
```

Vertex and fragment shaders for the Blinn-Phong lighting model.

High Performance Made Simple

The RapidMind Development Platform allows developers to use standard C++ programming to easily create applications targeted for high performance processors, including the Cell BE, GPUs, and CPUs. In the case of GPUs, the RapidMind platform implements user-specified arbitrary computations on the GPU, without any explicit reference by the developer to graphics APIs. The RapidMind platform provides a simple computational model that can be targeted by programmers and then maps this model onto any available computational resources in a system. Code can be written once, then run in parallel on any of the processors that the RapidMind Development Platform supports.

About RapidMind Inc.

RapidMind provides a software development platform that allows applications to take advantage of a new generation of high performance processors, including the Cell BE, GPUs, and other multi-core processors. The RapidMind Development Platform enables applications to realize the performance breakthroughs offered by these processors. Based in Waterloo, Canada, RapidMind is a venture-backed private company that is built on over five years of advanced research and development.

For more information on how RapidMind can help you with your high performance application, visit **WWW.RAPIDMIND.NET**

060320