The RapidMind Development Platform and Data-Parallel Programming

Michael McCool
Agenda

• Company background
• Product overview
• Basic tutorial
• Loop conversion example
• Application examples
Company

- Technology based on over five years of research at the University of Waterloo
  - One of first research labs to attempt GP-GPU in 1999
  - Developed language prototypes on simulator in 2001
  - Original publications in 2002 on metaprogrammed interface
- Company incorporated in June 2004
- VC Financing totaling $11.3M to date
- Product targets “High-Productivity Computing”
  - Easy to use, high performance, portable SW development
  - Targets GPUs and Cell BE
  - Extension to multi-core CPU now under development
• RapidMind Platform
  – *Programming middleware for many-core processors*
  – Single-source solution for portable parallel programming
  – Safe and deterministic data-parallel programming model
  – Scalable to arbitrary number of cores
  – Integrates with existing C++ compilers

• Write once, run on multiple targets
  – NVIDIA GPUs
  – ATI GPUs
  – Cell BE
  – Prototype demonstration on quad-core CPU
Product Benefits

- **Programmability**
  - No new tools or workflows
  - No need for low-level understanding of the target device
  - General purpose

- **Portability**
  - Application programming independent of OS or target device

- **Performance**
  - Automatically leverages *all* available computational resources
  - Optimizes code using dynamic runtime compilation
  - C++ overhead “compiled out” of generated code
  - Can significantly *outperform* native tools
• Use *existing* ISO standard C++ compiler:
  – Just include a header file, link to a library
  – Single-source solution, can be used with existing code bases
  – Does *not* require modification of debugging and build environments

• Allows specification of *arbitrary computation*:
  – *NOT* just a library of canned functions
  – Uses its own runtime optimizing code generator
  – User can specify *arbitrary* computational kernels
  – Staged compilation strategy avoids overhead of C++
Portability

• Multiple hardware targets:
  – NVIDIA GPUs
  – AMD/ATI GPUs
  – Cell BE
  – Prototype for x86 multi-core demonstrated

• Independent of number of cores

• Independent of memory model
  – Shared or distributed

• Can support new co-processor or accelerator without even recompiling program!
Cell BE Performance

- QJulia application
- Compared with previous IBM SDK implementation
- IBM implementation released over several months
- RapidMind implementation done in a few hours
- **Comparable or superior performance to IBM SDK implementation**
- **Overall code size and complexity significantly lower than that of IBM SDK implementation**
- **RapidMind version is portable to other processors**
GPU Performance

- Financial quasi Monte-Carlo option-pricing benchmark done in “competition” with HP
- CPU code uses icc autovectorization; tuned by HP, running on single Woodcrest core
- GPU implementation on an NVIDIA 7900GTX
- **GPU implementation over 32x faster than single-core CPU implementation**
CPU Performance

- Same financial quasi Monte-Carlo option-pricing benchmark as for GPU benchmark
- RapidMind implementation basically the same as the GPU implementation
- Prototype backend targeting Intel quad-core
- *RapidMind over 2x faster on one core, 8x faster on four cores*
A C++ Library API
  - For specifying arbitrary parallel computation

A parallel programming language
  - Embedded inside C++

Vocabulary for parallel programming
  - Set of nouns (types) and verbs (operations)
  - Added to existing standard language: ISO C++
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container for fixed-length data</td>
<td>Value</td>
</tr>
<tr>
<td>Container for variable-sized multidimensional data</td>
<td>Array</td>
</tr>
<tr>
<td>Container for computations</td>
<td>Program</td>
</tr>
</tbody>
</table>
Values

1 half
2 double
Value<3, float>
4 int

Tuple size
Element type
Values

 Tuple size

 1h
 2d
 Value 3f
 4i

 Element type
• Operators:
  +, -, *, /, %, &, |, ^, <, . . .
• Swizziling and writemasking:
  `Value4f c;`
  `c(2,1,0)`
  `c(0,0,0)`
  `c(1,1,2,3)`
  `c[3]`
Verbs: Functions

- Can declare functions in the usual way:

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

- Standard library
  - Matrix operations
  - Geometric operations
  - Trigonometry
  - Exponentials and logarithms
  - Splines, interpolation, and polynomials
  - etc.
Arrays

1 Value4d
Array<2,Value3f>
3 Value2i
Array Semantics

- Arrays use by-value semantics
  - Can assign arrays with O(1) cost
  - Strong modularity
  - Simple and easy to understand
  - Avoids needs for pointers to arrays
  - Consistent with value tuples

- Most data copies are optimized away
  - Exploits parallel assignment semantics

- By-reference semantics available:
  - **Accessor** type reference or “view” of region
Program Definition

Program p;

p = BEGIN {
In<Value3f> a, b;
Out<Value3f> c;
Value3f d = f(a, b);
c = d + a * 2.0f;
} END;
• Apply programs to arrays, get new arrays

\[ C = p(A, B) ; \]

*Invokes parallel execution*
Apply functions to arrays:
- Application: \( C = f(A, B) \)
- May have control flow (SPMD model)
- May perform random reads from other arrays
- Can read and write to subarrays

Apply collective operations to arrays:
- Reduce: \( a = \text{reduce}(p, A) \)
- Gather: \( A = B[U] \)
- Scatter: \( A[U] = B \)
- Others…
Program p;

p = BEGIN {
   In<Value3f> a, b;
   Out<Value3f> c;

   Value3f d = f(a, b);
   IF (all(a > 0.0f)) {
      c = d + a * 2.0f;
   } ELSE {
      c = d - a * 2.0f;
   } ENDIF;
} END;
SIMD:
- *Single Instruction, Multiple Data*
- Kernels include sequences of simple instructions
- Take constant amount of time to execute

SPMD:
- *Single Program, Multiple Data*
- Kernels may include control flow (loops and conditionals)
- Can avoid unnecessary work

SPMD includes but is *intrinsically* more powerful than SIMD
SIMD scheduling
• Assumes constant time per kernel

SPMD scheduling
• Takes variable execution time into account
• Load balancing distributes workload evenly across cores
Conversion:

1. Replace Types
2. Capture Computation
3. Parallel Execution
#include <cmath>

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

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] = f *
            (a[y][x][k] + b[y][x][k]);
        }
    }
}
```cpp
#include <cmath>

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

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] = f * 
                (a[y][x][k] + b[y][x][k]);
        }
    }
}
```

```cpp
#include <rapidmind/platform.hpp>
using namespace rapidmind;

Value1f f;
Array<2,Value3f> a(512,512);
Array<2,Value3f> b(512,512);
```
```cpp
#include <cmath>

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

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] = f *
            (a[y][x][k] + b[y][x][k]);
        }
    }
}
```

```cpp
#include <rapidmind/platform.hpp>
using namespace rapidmind;

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

Program prog = BEGIN {
    In<Value3f> r, s;
    Out<Value3f> q;
    q = f * (r + s);
} END;
```
```cpp
#include <cmath>
float f;
float a[512][512][3];
float b[512][512][3];

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] = f * (a[y][x][k] + b[y][x][k]);
        }
    }
}

#include <rapidmind/platform.hpp>
using namespace rapidmind;

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

Program prog = BEGIN {
    In<Value3f> r, s;
    Out<Value3f> q;
    q = f * (r + s);
} END;

a = prog(a,b);
```
Usage:
- Include platform header
- Link to runtime library

Data:
- Tuples
- Arrays
  - Global data abstraction

Programs:
- Defined dynamically
- Safe parallel execution
  - Remote procedure abstraction

```cpp
#include <rapidmind/platform.hpp>
using namespace rapidmind;

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

Program prog = BEGIN {
  In<Valuelf> r, s;
  Out<Valuelf> q;
  q = f * (r + s);
} END;

a = prog(a,b);
```
• Can just use existing IDE
• Single-step through code in immediate mode
• RapidMind control flow DOES work outside of program definitions (in “immediate mode”)
• Program objects can provide reports on performance
• Program objects can output optimized, annotated code
• Mechanisms available to attach annotations to code
• Synchronization mechanism for accurate timing
• Compilation logs can warn of use of features that are inefficient for particular hardware targets
Summary

• Provides abstractions for both code and data
  – Use C++ modularity, but compile out overhead
• Multiple hardware targets
  – GPU
  – Cell BE
  – Multi-core CPUs demonstrated and under development
• Simple, safe programming model
  – Avoids race conditions, deadlock, non-determinism
• Can achieve outstanding performance and productivity
  – Minimize training, effort, and risk
  – Maximize return on investment
• Single-source ISO standard C++ program:
  – No extensions or preprocessor needed
  – Works with existing compilers
Examples

- Crowd simulation
- Ray tracing (by RTT, in shipping product)
- Fast Fourier transform
- Convolution
- Quasi Monte Carlo option pricing
- Matrix-matrix multiply (SGEMM)
- Transformation and lighting
- Color and gamma correction
- Object tracking
- Sorting
- Set intersection (used for keyword search)
- Deferred shading
- Solution of partial differential equations
- Others…
Crowd Simulation
• Graphics on GPU
  – Shaders *also* implemented using RapidMind platform

• Behavioral Simulation on Cell BE Blade
  – 16K autonomous characters (4K visible at once)

• Parallel Execution:
  – Rules to simulate social behavior and basic physics

• Global Communication:
  – Any character can interact with any other
    • Requires (approximate) solution to K-nearest-neighbor problem
  – Behavior depends on the environment
    • Random access to environmental parameter grid
    • Obstacles, ground cover and slope
• Fundamental signal processing operation
  – Image processing
  – Pattern matching
  – Solving differential equations

• Standard test case for parallel computation

• Involves both
  – Computation
  – Communication

• Many varieties and ways to implement
  – Will show radix-2 split-stream complex-to-complex 1D FFT
// Fast Fourier Transform
Array<1,Value2f>
FFT (Array<1,Value2f> data, int n) {
    int N = (1 << n);

    // define program objects
    ...

    // generate and scramble twiddle factors with gather
    ...

    // scramble input data using a gather
    ...

    // perform split-stream FFT using lg(N) passes
    ...
}


// define program objects
Program butterfly_A = BEGIN {
    In<Value2f> a, b;
    Out<Value2f> c = a + b;
} END;

Program butterfly_B = BEGIN {
    In<Value2f> a, b, w;
    Value2f t = a - b;
    Out<Value2f> c;
    c[0] = t[0]*w[0] + t[1]*w[1];
    c[1] = t[1]*w[0] - t[0]*w[1];
} END;
// generate and scramble twiddle factors with gather
Array<1,Value2f> w(N/2);
w = twiddle(n-1)[ bitreverse(n-1) ];

// allocate temporary storage
Array<1,Value2f> x[2];
x[0] = Array1D<Value2f>(N);
x[1] = Array1D<Value2f>(N);

// scramble input data using a gather
x[0] = data[ bitreverse(n) ];

// initialize source marker
int src = 0;
// perform split-stream FFT using log(N) passes
for (int k=n-1; k>=0; k--) {
    // write into lower half of output array
    take(x[!src],N/2) = butterfly_A(
        stride(x[src],2),
        stride(offset(x[src],1),2)
    );
    // write into upper half of output array
    offset(x[!src],N/2) = butterfly_B(
        stride(x[src],2),
        stride(offset(x[src],1),2),
        take(w,1<<k)
    );
    // swap source and destination buffers
    src = !src;
}
// return final transform
return x[src];
Convolution

- Fundamental signal processing operation
- For large filters, use FFT
  - FFT
  - Elementwise complex multiplication
  - Inverse FFT
- For small filters, do directly
  - Shift flipped filter to each pixel, multiply, sum
  - May process many images with one filter
  - Filters used in pattern matching may be sparse
  - Can exploit sparsity to get more efficient execution
Convolution

Confocal microscopy image courtesy of Peter J. Lu, Harvard
float filter[N0][N1];
Array<2,Value1f> image(M0,M1);

Program convolve = BEGIN {
    In<Value2i> u;
    Out<Value1f> result = Value1f(0.0f);
    for (int i = 0; i < N0; i++) {
        for (int j = 0; j < N1; j++) {
            if (filter[i][j] != 0.0f) {
                Value2i tap = u - Value2i(i,j);
                result += filter[i][j] * image[tap];
            }
        }
    }
} END;

image = convolve(grid(M0,M1));
• **Real-time raytracing**
  – Supports reflection and refraction
  – Many recursive rays per pixel
  – Approximately 15Hz on real CAD data

• **Commercial product:**
  – Developed by RTT AG, Germany
  – Used for automotive CAD visualization
  – *Shipping product*

• **Hardware:**
  – Released product runs on NVIDIA GPUs on HP workstations
  – Demonstrated on Cell BE at SIGGRAPH 2006
Raytracing