The RapidMind Platform for High-Performance, High-Productivity Parallel Programming

Michael McCool
Chief Scientist, RapidMind
Georgia Tech Cell Workshop
11 July 2008
Performance
• *Fully* leverage the potential of *all* processors/cores
• Support for adaptive programs and auto-tuning

Portability
• Hardware independent applications
• Automatic scalability to additional cores
• Exploit future many-core processors efficiently
• Use accelerators when available

Programmability
• Simple extensions of existing practice
• Automate, but provide drill-down when needed
• Explicit control of important policy decisions
Programming the Cell processor with RapidMind provides equivalent performance with significantly reduced effort.

Cell Performance
- IBM’s benchmark application
- RapidMind and Cell SDK
  vs
  Best known implementation on Cell SDK alone
- **RapidMind nearly doubled performance with significantly less developer effort**
Deployment choice now, but also future proof
Database transactions  
Image/video manipulation  
Enterprise search  
Data mining  
Real-time data analysis  
3D visualization  
Crowd simulation  
Convolution  
Electromagnetic simulation  
Seismic analysis  
Ray tracing (w/ RTT)  
Fast Fourier transform  
Monte Carlo option pricing  
Binomial tree option pricing  
Linear algebra  
Transformation and lighting  
Color and gamma correction  
Anisotropic diffusion filtering  
Fluid flow  
Motion estimation  
Collision detection  
Rigid body simulation  

Broadcast-quality encoding  
Medical imaging  
Film and television content generation  
Image and signal processing  
Financial analysis  
Seismic analysis  
Object tracking  
Sorting  
Quaternion Julia set  
Deferred shading/relighting  
Vector (SVG, PDF) textures  
Advanced reflectance models  
K nearest neighbour search  
Keyword search  
N-body force computation  
Differential equation solver  
Optimization under constraints  
Tomographic reconstruction  
Volume rendering  
Time series search  
EM time series forecasting  
Pattern recognition  

Many others...
Raytracing

Real-time raytracing
Reflection and refraction
Many recursive rays per pixel
Incoherent memory access
Accelerator traversal
Runs on GPU and Cell

Commercial product:
Developed by RTT AG, Germany
Automotive CAD visualization
The RapidMind Platform
Overview

- **Software development platform for multi-core and many-core processors**
- Single-source solution for portable high-performance parallel programming
- Supports high productivity development
- Safe and deterministic general-purpose programming model (SPMD stream)
- Scalable to an arbitrary number of cores
- Can be used to target both accelerators and multi-core processors, including Cell
- Integrates with existing C++ compilers
A good programming technology should:

1. Provide an accurate conceptual model of the hardware
2. Clearly expose the most important policy decisions and architectural elements of the hardware
3. Provide structure and modularity
4. Automate what can be automated, and not overload the programmer with trivia
5. Provide drill-down mechanisms for use when necessary
Key Policy Factors for Multi-core Performance

1. Parallelism
2. Memory Locality

- Want to expose these at highest level of programming model
- Rest is details, let automation handle (in most cases)
1. Parallelism
   - Choose or design a good parallel algorithm
   - Must scale with additional processing elements
   - Needs to be independent of number of cores

2. Memory Locality
   - Efficient use of the memory hierarchy
     - More frequent use of faster local memory
   - Planned, coherent use of memory and data transfer
   - High arithmetic intensity
   - Clear expression of locality
   - Clear expression of data movement
Apply functions to arrays:
- Application: $C = f(A, B)$
- May have control flow (SPMD model)
- May have local arrays
- May call other functions
- May access other arrays
- Can read and write to subarrays

Use collective operations:
- Reduce: $a = \text{reduce}(f, A)$
- Gather: $A = B[U]$
- Scatter: $A[U] = B$
- Others . . .
Advantages of SPMD Stream Processing

• Efficient on a variety of architectures
  • Shared memory machines
  • Distributed memory machines
  • Vector/stream machines
  • SIMD-within-a-register/multi-core machines

• Predictable memory access patterns

• Scales to any number of cores

• Single conceptual thread of control
  • Simple extension of existing programming practice
  • No explicit synchronization needed
  • No deadlocks or race conditions
  • Debugging simplified
API
- Integrates with C++
- Requires no new tools or workflow

Platform
- **Code Optimizer** analyzes and optimizes computations to remove overhead
- **Load Balancer** plans and synchronizes work to keep all cores fully utilized
- **Data Manager** reduces data bottlenecks
- **Logging/Diagnostics** detects and reports performance bottlenecks

Processor Support Modules
- ATI/AMD and NVIDIA GPUs
- Cell Blade, Cell Accelerator Board, PS3
- AMD and Intel x86 Multi-core CPUs
Library:
- Use of “canned” functions tuned for performance
- Suitable for some problems, but inflexible

New Language or Language Extensions:
- Major training and implementation hurdle
- Build systems, IDEs, adoption, supported platforms, etc. etc.

Platform:
- *Embedded programming interface*
- Use like a library:
  - Include header file, link, use existing compiler, IDEs, build system, etc.
- As expressive as a language
- Allows adaptive, auto-tuned software development
Computation Capture and Transformation Process

- Interface extracts computation expressed in C++ while eliminating overhead
- Code generator creates native machine code
- Runtime tightly couples multiple optimizations and manages execution over multiple cores

Standard C++ using RapidMind interface

C++ source code

Standard C++ Tools

RapidMind Collection

Massively parallel computation

RapidMind Compilation

Platform specific code

RapidMind Execution

Streaming execution

Multicore Processor

Standard executable with embedded RapidMind operations
<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

1h
2d
Value 3f
4i

Tuple size
Element type
Arrays

1 Value4d

Array<2,Value3f>

3 Value2i
Program p;

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

Value3f d = f(a, b);
c = d + a * 2.0f;
} 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
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;
• Apply programs to arrays, get new arrays

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

*Invokes parallel execution asynchronously*

*Computation only forced to completion when C read*
• Apply programs to arrays, get new arrays
  \[ C = p(A,B); \]

• Can use accessors to get subarrays
  \[ \text{slice}(C,500,999) = p(\text{slice}(A,0,499),\text{take}(B,500)); \]

• Can use programs as parameters to collectives
  \[ a = \text{reduce}(p,B); \]
Usage:
- Include platform header
- Link to runtime library

Data:
- Tuples
- Arrays
- Global data abstraction

Programs:
- Defined dynamically
- Execute on coprocessors
- Dynamic remote procedure abstraction

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

Valuelf f = 2.0f;
Array<2,Valuel3f> a(512,512);
Array<2,Valuel3f> b(512,512);

Program prog = BEGIN {
  In<Valuel3f> r, s;
  Out<Valuel3f> q;
  q = (r + s) * f;
} END;

a = prog(a,b);
f = 3.0f;
stride(a,2,2) = prog(
  slice(a,0,255,0,255),
  slice(b,256,511,0,255));
Programming the Cell processor with RapidMind provides equivalent performance with significantly reduced effort.

**Cell Performance**
- IBM’s benchmark application
- RapidMind and Cell SDK
  vs
  Best known implementation on Cell SDK alone
- **RapidMind nearly doubled performance with significantly less developer effort**
Program m_program = BEGIN {
    In<Value2i> index;   // pixel coordinates
    Out<Value3f> colour = m_bgcolour;  // pixel colour

    // ... ‘rD’ is a function of ‘index’
    Value1f dist = intersectQJulia(rO, rD, m_mu,
                                   max_iterations, m_epsilon);

    IF (dist < m_epsilon + .01f) {
        Value3f N = normEstimate (rO, m_mu,
                                   max_iterations);

        colour = Phong (m_light, rD, rO, N);
        //...
    } ENDIF;
} END;
Comparison with Cell Intrinsics

```c
static inline void IterateIntersect (vector float *_q, vector float *_qp, 
    vector float c, int maxIterations) {
    vector float q = *_q;
    vector float qp = *_qp;
    int i;
    for (i = 0; i < maxIterations; i++) {
        qp = spu_mul (VEC_LITERAL (vector float, 2.0f), quatMult (q, qp));
        q = spu_add (quatSq (q), c);
        if (dot4 (q, q) > ESCAPE_THRESHOLD) break;
    }
    *_qp = qp;
    *_q = q;
}

void iterateIntersect (Value4f& q, Value4f& qp, Value4f c, 
    int maxIterations) {
    Value1i i;
    FOR (i = 0, i < maxIterations, i += 1) {
        qp = Value1f(2.0) * quatMult(q, qp);
        q = quatSq(q) + c;
        BREAK (dot(q, q) > ESCAPE_THRESHOLD);
    } ENDFOR;
```

Copyright © 2008 RapidMind Inc. All rights reserved.
Comparison with Vectorized

```
static inline void IterateIntersectT (vector float *_q, vector float *_qp, vector float *c, int maxIterations)
{
    int i;
    vector float dq;
    vector float qx = _q[0];  
    vector float qy = _q[1];  
    vector float qz = _q[2];  
    vector float qw = _q[3];  
    vector float qpx = _qp[0]; 
    vector float qpy = _qp[1]; 
    vector float qpz = _qp[2]; 
    vector float qpw = _qp[3]; 
    vector unsigned int vbbreak, sbbreak;
    const vector float vtwo = VEC_LITERAL (vector float, 2.0f);
    const vector float vESC = VEC_LITERAL (vector float, ESCAPE_THRESHOLD);
    vector unsigned int write_mask = VEC_LITERAL (vector unsigned int, 0xFFFFFFFF);
    for (i = 0; i < maxIterations; i++) {
        quatMult4 (qx, qy, qz, qw, qpx, qpy, qpz, qpw, &qpx, &qpy, &qpz, &qpw);
        qpx = spu_mul (vtwo, qpx);
        qpy = spu_mul (vtwo, qpy);
        qpz = spu_mul (vtwo, qpz);
        quatSq4 (qx, qy, qz, qw, &qx, &qy, &qz, &qw);
        qx = spu_add (qx, c[0]);
        qy = spu_add (qy, c[1]);
        qz = spu_add (qs, c[2]);
        qw = spu_add (qw, c[3]);
        dq = _dot_product4v (qx, qy, qz, qw, vbreak);
        vbbreak = spu_cmpgt (dq, vESC);
        _q[0] = spu_sel (_q[0], q, write_mask);
        _q[1] = spu_sel (_q[1], qy, write_mask);
        _q[2] = spu_sel (_q[2], qz, write_mask);
        _q[3] = spu_sel (_q[3], qw, write_mask);
        _qp[0] = spu_sel (_qp[0], qpx, write_mask);
        _qp[1] = spu_sel (_qp[1], qpy, write_mask);
        _qp[2] = spu_sel (_qp[2], qpz, write_mask);
        _qp[3] = spu_sel (_qp[3], qpw, write_mask);
        write_mask = spu_andc (write_mask, vbreak);
        sbbreak = spu_gather (write_mask);
        if (spu_extract (sbbreak, 0) == 0) break;
    }
}
```

```
void iterateIntersect (Wide<Value4f>& q, 
        Wide<Value4f>& q, 
        const Wide<Value4f>& c, 
        int maxIterations) 
{
    Value1i i;
    FOR (i = 0, i < maxIterations, i += 1) { 
        qp = Value4f(2,2,2,2) * quatMult(q, qp);
        q = quatSq(q) + c;
        BREAK (all(dot( q, q ) > ESCAPE_THRESHOLD));
    } ENDFOR;
}
```
Main SPU Loop:
Elimination of Glue Code

```c
int main (int spuid, addr64 args, addr64 env)
{
    unsigned int i, j;  unsigned int ea_low;
    unsigned int opcode;
    unsigned int now_tag = 2;  unsigned int now_mask = 1 << now_tag;
    unsigned int wb_addr = NULL;
    vector float c = VEC_LITERAL (vector float, 0.0f);
    vector float light = VEC_LITERAL (vector float, 0.0f);
    float epsilon = 0;  int renderShadows = 0;  int maxIterations = 0;
    vector float r_top_start;  vector float r_bottom_start;
    vector float r_bottom_stop;  vector float dir_top;  vector float dir_bottom;
    vector float curr_dir;  vector float curr_dir2;
    int curr_buf = 0;  unsigned int fb_ea_low;  unsigned int fb_row_stride;  unsigned int fb_store;
    unsigned int num_columns;  unsigned int start_column;
    unsigned int num_rows;  float imgWidth = 0, imgHeight = 0;
    vector float px, py;  vector float py2;
    vector float ddx, ddy;
    while (1)
    {
        opcode = (unsigned int) spu_read_in_mbox ();
        switch (opcode)
        {
        case FLUSH_SPE_OP:
            mywriteback[0] = BUFFER_READY;
            mfc_putchar[0] (mywriteback, wb_addr, 16, now_tag, 0, 0);
            mfc_write_tag_mask (now_mask | sc_mask[0] | sc_mask[1]);
            mfc_read_tag_status_all ();
            break;
        case EXIT_SPE_OP:
            return 0;
            break;
        case RENDER_CONTEXT_SPE_OP:
            ea_low = (unsigned int) spu_read_in_mbox ();
            mfc_putchar (void *) &rc, ea_low, sizeof rendering_context, now_tag, 0, 0);
            mfc_write_tag_mask (now_mask | sc_mask[0] | sc_mask[1]);
            mfc_read_tag_status_all ();
            rO = *((vector float *) rc.eyeP);
            eye = *((vector float *) rc.eyeP);
            light = *((vector float *) rc.lightP);
            epsilon = rc.epsilonP;
            renderShadows = rc.shadowsP;
            maxIterations = rc.iterationsP;
            wb_addr = (unsigned int) rc.wb_addr;
            imgWidth = (float) rc.img_width;
            imgHeight = (float) rc.img_height;
            break;
        case RAY_REGION_SPE_OP:
            ea_low = (unsigned int) spu_read_in_mbox ();
            mfc_putchar (void *) &region, ea_low, sizeof rendering_region, now_tag, 0, 0);
            mfc_write_tag_mask (now_mask | sc_mask[0] | sc_mask[1]);
            mfc_read_tag_status_all ();
            mu = *((vector float *) region.muP);
            r_top_start = *((vector float *) region.r_top_start);
            r_bottom_start = *((vector float *) region.r_bottom_start);
            r_bottom_stop = *((vector float *) region.r_bottom_stop);
            fb_ea_low = region.fb_ea_low;
            fb_row_stride = MAX_IMG_HEIGHT * sizeof (vector float);
            num_rows = imgHeight;
            num_columns = region.column_count;
            start_column = region.start_column;
            fb_store = fb_ea_low + (start_column * fb_row_stride);
            dir_top = r_top_start;
            dir_bottom = r_bottom_start;
            inv_width = _inverse_v (spu_splats (imgWidth));
            inv_height = _inverse_v (spu_splats (imgHeight));
            ddx = spu_sub (r_bottom_stop, r_bottom_start);  ddy = spu_sub (r_top_start, r_bottom_start);
            ddx = spu_mul (ddx, inv_width);
            ddy = spu_mul (ddy, inv_height);
            px = spu_splats ((float) start_column);
            for (i = 0; i < num_columns; i++)
            {
                py = VEC_LITERAL (vector float, 0.0f);
                py2 = VEC_LITERAL (vector float, 1.0f);
                mfc_write_tag_mask (sc_mask[i]);
                mfc_read_tag_status_all ();
                case FLUSH_SPE_OP:
                    mywriteback[0] = BUFFER_READY;
                    mfc_putchar[0] (mywriteback, wb_addr, 16, now_tag, 0, 0);
                    mfc_write_tag_mask (now_mask | sc_mask[0] | sc_mask[1]);
                    mfc_read_tag_status_all ();
                    break;
                case EXIT_SPE_OP:
                    return 0;
                    break;
                case RENDER_CONTEXT_SPE_OP:
                    ea_low = (unsigned int) spu_read_in_mbox ();
                    mfc_putchar ((void *) &rc, ea_low, sizeof rendering_context, now_tag, 0, 0);
                    mfc_write_tag_mask (now_mask | sc_mask[0] | sc_mask[1]);
                    mfc_read_tag_status_all ();
                    rO = *((vector float *) rc.eyeP);
                    eye = *((vector float *) rc.eyeP);
                    light = *((vector float *) rc.lightP);
                    epsilon = rc.epsilonP;
                    renderShadows = rc.shadowsP;
                    maxIterations = rc.iterationsP;
                    wb_addr = (unsigned int) rc.wb_addr;
                    imgWidth = (float) rc.img_width;
                    imgHeight = (float) rc.img_height;
                    break;
            }
        
        m_colours = m_program(grid(size,size));
```
Programming the Cell BE can be easy

- Single-source solution based on embedded interface
- Get modularity and structure of C++ without overhead
- Capable of same performance as low-level programming

Programming models are important!

- Data-driven parallelism has scalability advantages
- SPMD Stream Programming model
- Intrinsically safe, but drill-downs available when needed

Programming platforms:

- *Not* necessary to introduce completely new languages
- Can obtain similar performance and expressiveness within *standard* C++ and *existing* compilers