Using the RapidMind™ Development Platform, porting to and tuning for the Cell Broadband Engine™ (Cell BE) processor is much simpler than doing so by hand and therefore can provide better performance at less effort than other methods while maintaining portability and programmability.

Introduction
The RapidMind Development Platform is a software development system that allows the use of standard C++ to easily create high-performance and massively parallel applications that run on parallel architectures such as the Cell Broadband Engine™ (Cell BE) processor, Graphics Processing Units (GPUs) and multi-core CPUs.

The Cell BE processor contains nine cores in total. One of these cores, the PowerPC Processing Unit (PPU), uses a traditional CPU architecture. The other eight cores are specialized vector processors known as Synergistic Processing Units (SPUs). To achieve high performance on the Cell BE it is critical to target these SPU cores. Without RapidMind, this involves writing very low level code due to the specialized nature of these processing units. RapidMind makes it possible to write platform-independent high level code in C++ which still takes full advantage of all cores in the Cell Broadband Engine™ processor.

This case study aims to demonstrate the simplicity of porting an existing application and tuning it with the RapidMind Development Platform. It is critical to tune the application to achieve high performance. Using RapidMind, this tuning can be done while maintaining portability. We demonstrate this on an independently written program, porting it from Cg/C++/OpenGL to C++ and RapidMind. The program chosen is a Quaternion Julia Set renderer.

Initial Port
The original rendering source code was written using Cg and C++. Converting the application to use the RapidMind platform was mostly trivial. The Cg code represents the core parallel algorithm used to generate the image. This code was ported to C++ code using RapidMind's types and operations. The rest of the original program, written in C++, mostly consisted of the wrapper used performance. Using RapidMind, this tuning can be done while maintaining portability. We demonstrate this on an independently written program, porting it from Cg/C++/OpenGL to C++ and RapidMind. The program chosen is a Quaternion Julia Set renderer.

The same algorithm implemented using hand-written C and SPU intrinsics above, compared to RapidMind-enabled C++ code below. Note the simplicity and generality of the RapidMind version.

```
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);
qpw = spu_mul (vtwo, qpw);
quatSq4 (qx, qy, qz, qw, &qx, &qy, &qz, &qw);
qx = spu_add (qx, c[0]);
qy = spu_add (qy, c[1]);
qz = spu_add (qz, c[2]);
qw = spu_add (qw, c[3]);
```

```
qp = cond(done, qp, Wide<Value1f>(2.0f) * quatMult(q, qp));
q = cond(done, q, quatSq(q) + c);
done = dot(q, q) > ESCAPE_THRESHOLD;
```
to execute the Cg code and incorporated use of OpenGL. This code was mostly unnecessary in the RapidMind version since the platform takes care of the setup and execution of parallel algorithms, whether on the GPU or on the Cell BE.

Tuning
The application was then tuned to achieve better performance on the Cell BE processor. These tuning steps were generally straightforward and involved no substantial increase in the code base. Two tuning steps had the largest impacts on performance. In the first step, the code was vectorized to operate on four pixels of the result image at a time, allowing better use of the Cell’s SIMD instructions. By using C++’s modularity combined with RapidMind’s types, this step left the structure and size of the code largely untouched. The other major step involved partial and full unrolling of some of the inner loops of the application. Because of the relationship between RapidMind and C++, this step was trivial to perform.

Results and Comparison
The performance results were then compared to an IBM implementation of the same application using SPU intrinsics. Neither the IBM nor the RapidMind implementations were exhaustively tuned. However, it was clear that the RapidMind implementation was far simpler to implement and tune, allowing it to reach close to twice the performance of the IBM implementation while being simpler and more general.

The following page presents performance comparisons between the IBM implementation using C and SPU intrinsics, and the RapidMind implementation using C++ and the RapidMind Development Platform. Code comparisons are also given in the subsequent pages.

IBM has indicated to us that they expect the C/SPU intrinsics implementation to gain performance by using the XLC compiler instead of GCC, but numbers were unavailable at the time of writing.

Conclusion
Using RapidMind, application developers can not only move their applications to parallel processors such as the Cell BE with ease, they are also able to perform tuning to adapt their applications to particular processors or environments much more simply than when using lower-level techniques such as assembly or intrinsics. The platform allows taking advantage of these processors without introducing a performance overhead.

Acknowledgements
We would like to acknowledge Barry Minor from IBM Corporation for the C/SPU intrinsics implementation of the application, as well as his helpful comments and validation of the comparison results.

This product includes software developed by Keenan Crane. His original Cg quaternion julia set renderer is available at http://graphics.cs.uiuc.edu/svn/kcrane/web/project_qjulia.html.

RapidMind provides a software development platform that allows software vendors to deliver high performance on multi-core and stream processors, including the GPU and the Cell BE. Without sacrificing development simplicity, RapidMind-enabled applications experience a dramatic leap in performance.

For more information on RapidMind, visit our web site at http://www.rapidmind.net/.
Performance comparison between IBM and RapidMind implementation at various tuning stages. Frame rate for a single 1024x1024 frame with 4 Julia set iterations is given. IBM numbers generated using IBM sample code compiled with gcc.

Performance scaling and efficiency of automatic RapidMind load balancing. The chart shows that as the number of SPUs used increases, the performance scales accordingly, with scaling efficiency above 95% for 8 SPUs, and above 87% for 16 SPUs using two Cell Broadband Engine processors in an IBM QS20 BladeCenter® server.
void iterateIntersect(inout float4 q, inout float4 qp, float4 c, int maxIterations)
{
    for(int i=0; i<maxIterations; i++) {
        qp = 2.0 * quatMult(q, qp);
        q = quatSq(q) + c;

        if (dot(q, q) > ESCAPE_THRESHOLD) break;
    }
}

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, float4 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;
}

Comparison of a single function from the application in Cg (the original source code), C with SPU intrinsics (IBM implementation) and C++ with the RapidMind Development Platform.
static inline void IterateIntersect(vector float _q, vector float _qp, vector float _c, int maxIterations)
{
    int i;
    vector float dqq;
    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 vbreak, sbreak;
    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);
        qpw = spu_mul(vtwo, qpw);
        quatSq4(qx, qy, qz, qw, &qx, &qy, &qz, &qw);
        qx = spu_add(qx, _c[0]);
        qy = spu_add(qy, _c[1]);
        qz = spu_add(qz, _c[2]);
        qw = spu_add(qw, _c[3]);
        dqq = _dot_product4_v(qx, qy, qz, qw, qx, qy, qz, qw);
        vbreak = spu_cmpgt(dqq, vESC);
        _q[0] = spu_sel(_q[0], qx, 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);
        sbreak = spu_gather(write_mask);
        if (spu_extract(sbreak, 0) == 0) break;
    }
}

void iterateIntersect(Wide<Value4f>& q, Wide<Value4f>& qp, const Wide<Value4f>& c, int maxIterations)
{
    Value1i i;
    Wide<Value1bool> done(false);
    FOR (i = 0, i < maxIterations, i += 1) {
        qp = cond(done, qp, Wide<Value1f>(2.0f) * quatMult(q, qp));
        q = cond(done, q, quatSq(q) + c);
        done = dot(q, q) > ESCAPE_THRESHOLD;
        BREAK (all(done));
    } ENDFOR;
}

Comparison of vectorized version of iterateIntersect operating on four pixels at once. The RapidMind code makes use of the Wide<> type modifier to express the vectorization, whereas the SPU intrinsic implementation is explicitly expanded. Note that the Wide<> type modifier is trivial to implement because RapidMind allows all of the modularity constructs in C++ to be used. These modularity constructs act as “scaffolding” which is torn out of the generated program. There is no performance overhead in using C++ classes, templates, structures, functions, etc.
Because RapidMind handles issues such as DMA for memory access and load balancing for efficient execution across multiple SPUs, executing code on the SPUs is trivial. Furthermore, because the entire system is programmed from a single-source C++ program, sharing variables between the Cell’s PPU and its SPUs does not require any additional “glue” code. The SPU intrinsics example does not show the corresponding PPU code necessary to set up and communicate with SPUs, which is also unnecessary when using RapidMind.

m_colours = m_program(grid(size, size));