Introducing Data Parallel C++: A Standards-based, Cross-Architecture Programming Language

Mike Kinsner
Software Engineer, Intel Corporation
Member, Khronos SYCL & OpenCL Working Groups
Introduce Data Parallel C++, the code structure, and key concepts to get you writing code quickly!
1. What is Data Parallel C++?
2. Program structure and execution model
3. Compiling code
4. Queues and device selection
5. Management of data and task graphs
6. Some DPC++ extensions
Programming Challenge

Diverse set of data-centric hardware

No common programming language or APIs

Inconsistent tool support across platforms

Each platform requires unique software investment
OneAPI for Cross-Architecture Performance

- Optimized Applications
- Optimized Middleware & Frameworks

OneAPI Product

- Direct Programming
  - Data Parallel C++
- API-Based Programming
  - Libraries
- Analysis & Debug Tools

Get functional quickly. Then analyze and tune.

Some capabilities may differ per architecture.
WHAT IS DATA PARALLEL C++?

Data Parallel C++

= C++ and SYCL* standard and extensions

Based on modern C++

▪ C++ productivity benefits and familiar constructs

Standards-based, cross-architecture

▪ Incorporates the SYCL standard for data parallelism and heterogeneous programming

Data Parallel C++ ⇔ DPC++
DPC++ Extends SYCL 1.2.1

Enhance Productivity
• Simple things should be simple to express
• Reduce verbosity and programmer burden

Enhance Performance
• Give programmers control over program execution
• Enable hardware-specific features

DPC++: Fast-moving open collaboration feeding into the SYCL standard
• Open source implementation with goal of upstream LLVM
• DPC++ extensions aim to become core SYCL, or Khronos extensions
A COMPLETE DPC++ PROGRAM

Single source
- Host code and heterogeneous accelerator kernels can be mixed in same source files

Familiar C++
- Library constructs add functionality, such as:

<table>
<thead>
<tr>
<th>Construct</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>Work targeting</td>
</tr>
<tr>
<td>buffer</td>
<td>Data management</td>
</tr>
<tr>
<td>parallel_for</td>
<td>Parallelism</td>
</tr>
</tbody>
</table>

```cpp
#include <CL/sycl.hpp>
#include <iostream>
constexpr int num=16;
using namespace cl::sycl;

int main() {
    auto R = range<1>{ num };
    buffer<int> A{ R };

    queue{}.submit([&](handler& h) {
        auto out =
            A.get_access<access::mode::write>(h);
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0];
        });
    });

    auto result =
        A.get_access<access::mode::read>();
    for (int i=0; i<num; ++i)
        std::cout << result[i] << "\n";

    return 0;
```
COMPILING A DPC++ PROGRAM

Use the Intel DPC++ compiler!

- dpcpp -fsycl-unnamed-lambda my_source.cpp –o executable

Finding the compiler:

Using Intel's oneAPI Beta
Test code and workloads across a range of Intel® data-centric architectures at
Intel® DevCloud for oneAPI
software.intel.com/devcloud/oneAPI

Learn more and download the beta toolkits at
software.intel.com/oneapi
oneAPI available now on INTEL DEVCLoud

A development sandbox to develop, test and run your workloads across a range of Intel CPUs, GPUs, and FPGAs using Intel’s oneAPI beta software

software.intel.com/en-us/devcloud/oneapi

Learn about oneAPI Toolkits
Learn Data Parallel C++
Evaluate Workloads
Build Heterogenous Applications
Prototype your project

NO DOWNLOADS | NO HARDWARE ACQUISITION | NO INSTALLATION | NO SET-UP AND CONFIGURATION
GET UP AND RUNNING IN SECONDS!
Kernels can be specified in multiple ways:

- C++ Lambdas
  - Often thin wrapper calling a function
- Functor Objects
- Interop

Kernels are submitted to queues:

- parallel_for
- single_task
- parallel_for_work_group
# Kernel-based Model

**Kernel**
- Code that executes on an accelerator, typically many times/instances per kernel invocation (across an ND-range)

**Kernels clearly identifiable in code**
- Small number of classes can define a kernel (e.g. `parallel_for`)

**Developer specifies where kernels will run**
- Varying levels of control

---

```cpp
#include <CL/sycl.hpp>
#include <iostream>
constexpr int num=16;
using namespace cl::sycl;

int main() {
    auto R = range<1>{ num };
    buffer<int> A{ R };

    queue{}.submit([&](handler& h) { auto out = A.get_access<access::mode::write>(h); h.parallel_for(R, [=](id<1> idx) { out[idx] = idx[0]; }); });

    auto result = A.get_access<access::mode::read>();
    for (int i=0; i<num; ++i)
        std::cout << result[i] << "\n";

    return 0;
}
```

---

Defines kernel
Data Parallelism is expressed using ND-Ranges

- Total Work = # Work-groups x # Work-items per Work-group
- Bottom-up, hierarchical single program multiple data (SPMD) model

Collective functions allow cross-work-item operations to be expressed (e.g. barrier). Write code from perspective of a single work-item.
## Choosing Where Device Kernels Run

### Work is submitted to queues

- Each queue is associated with exactly one device (e.g. a specific GPU or FPGA)
- You can:
  - Decide which device a queue is associated with (if you want)
  - Have as many queues as desired for dispatching work in heterogeneous systems

<table>
<thead>
<tr>
<th>Create queue targeting any device:</th>
<th>queue();</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create queue targeting a pre-configured classes of devices:</td>
<td>queue(cpu_selector{}); queue(gpu_selector{}); queue(intel::fpga_selector{}); queue(accelerator_selector{}); queue(host_selector{});</td>
</tr>
</tbody>
</table>
| Create queue targeting specific device (custom criteria): | class custom_selector : public device_selector {
  int operator()(...) {
      // Any logic you want!
  }
  ... queue(custom_selector{}); |

---

Always available

- Queue_A ➡️ GPU
- Queue_B ➡️ FPGA
- Queue_C ➡️ Always available
```cpp
int main() {
    auto R = range<1>{ num };  
    buffer<int> A[R], B[R];
    queue Q;

    Q.submit([&](handler& h) {
        auto out = A.get_access<access::mode::write>(h);
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0];
        });
    });

    Q.submit([&](handler& h) {
        auto out = B.get_access<access::mode::write>(h);
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0];
        });
    });

    Q.submit([&](handler& h) {
        auto in = A.get_access<access::mode::read>(h);
        auto inout = B.get_access<access::mode::read_write>(h);
        h.parallel_for(R, [=](id<1> idx) {
            inout[idx] *= in[idx];
        });
    });
}
```
**Buffers**: Encapsulate data in a SYCL application
- Across both devices and host!

**Accessors**: Mechanism to access buffer data
- Create data dependencies in the SYCL graph that order kernel executions

```cpp
int main() {
    auto R = range<1>{ num };
    buffer<int> A{ R }, B{ R };
    queue Q;

    Q.submit([&](handler& h) {
        auto out = A.get_access<access::mode::write>(h);
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0]; }); });

    Q.submit([&](handler& h) {
        auto out = A.get_access<access::mode::write>(h);
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0]; }); });

    // ...}
```
Think of a SYCL application as two parts:

1. Host code
2. The graph of kernel executions

These execute independently, except at synchronizing operations:

- The host code submits work to build the graph (and can do compute work itself)
- The graph of kernel executions and data movements executes asynchronously from host code, managed by the SYCL runtime
#include <CL/sycl.hpp>
#include <iostream>

constexpr int num = 16;
using namespace cl::sycl;

int main() {
    auto R = range<1>{ num };
    buffer<int> A{ R };

    queue{}.submit([&](handler& h) {
        auto out = A.get_access<access::mode::write>(h);
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0]; }); });

    auto result = A.get_access<access::mode::read>();
    for (int i = 0; i < num; ++i)
        std::cout << result[i] << "\n";

    return 0;
}
Reminder: DPC++

= ISO C++ and SYCL* standard and **extensions**

Here we’ll look at:

1. Unified Shared Memory
2. Sub-groups
3. Ordered queues
4. Pipes
5. Optional lambda name
UNIFIED SHARED MEMORY (USM)

The SYCL 1.2.1 standard provides a Buffer memory abstraction
• Powerful and elegantly expresses data dependences

However...
• Replacing all pointers and arrays with buffers in a C++ program can be a burden to programmers

USM provides a pointer-based alternative in DPC++
• Simplifies porting to an accelerator
• Gives programmers the desired level of control
• Complementary to buffers
## USM Allocations and Pointer Handles

### Allocation Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Accessibly By</th>
<th>Migratable To</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Device</strong></td>
<td><strong>Device allocation</strong></td>
<td>Host</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Device</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other device</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Host</strong></td>
<td><strong>Host allocation</strong></td>
<td>Host</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any device</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Shared</strong></td>
<td><strong>Potentially migrating allocation</strong></td>
<td>Host</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Device</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other device</td>
<td>?</td>
</tr>
</tbody>
</table>

Automatic data accessibility and explicit data movement supported

```c
auto A = (int*)malloc_shared(N*sizeof(int), ...);
auto B = (int*)malloc_shared(N*sizeof(int), ...);
...
q.submit([&](handler& h) {
    h.parallel_for(range<1>{N}, [=] (id<1> ID) {
        auto i = ID[0];
        A[i] *= B[i];
    });
});
```
BUFFERS ARE POWERFUL AND ELEGANT

- Use if the abstraction applies cleanly in your application, and/or buffers aren’t disruptive to your development.

USM PROVIDES A FAMILIAR POINTER-BASED C++ INTERFACE

- Useful when porting C++ code to DPC++, by minimizing changes.
- Use shared allocations when porting code, to get functional quickly.

USM NOT INTENDED TO PROVIDE PEAK PERFORMANCE OUT OF BOX

- Allows code to become functional quickly.
- Use profiler to identify small % of code where you should tune.
Declare C++ Arrays

```cpp
auto A = (int *) malloc(N * sizeof(int));
auto B = (int *) malloc(N * sizeof(int));
auto C = (int *) malloc(N * sizeof(int));

for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
}

buffer<int, 1> Ab(A, range<1>{N});
buffer<int, 1> Bb(B, range<1>{N});
buffer<int, 1> Cb(C, range<1>{N});

q.submit([&] (handler& h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=] (id<1> i) {
        aC[i] = aA[i] + aB[i];
    });
});
// A,B,C updated
```
BUFFER EXAMPLE – COMMON PATTERN

Declare C++ Arrays

```cpp
auto A = (int *) malloc(N * sizeof(int));
auto B = (int *) malloc(N * sizeof(int));
auto C = (int *) malloc(N * sizeof(int));
```

Initialize C++ Arrays

```cpp
for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
}
```

```cpp
buffer<int, 1> Ab(A, range<1>{N});
buffer<int, 1> Bb(B, range<1>{N});
buffer<int, 1> Cb(C, range<1>{N});
```

```cpp
q.submit([&] (handler& h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=] (id<1> i) {
        aC[i] = aA[i] + aB[i];
    });
});
```

// A,B,C updated
**BUFFER EXAMPLE – COMMON PATTERN**

1. **Declare C++ Arrays**
   ```cpp
   auto A = (int *) malloc(N * sizeof(int));
   auto B = (int *) malloc(N * sizeof(int));
   auto C = (int *) malloc(N * sizeof(int));
   ```

2. **Initialize C++ Arrays**
   ```cpp
   for (int i = 0; i < N; i++) {
       A[i] = i; B[i] = 2*i;
   }
   ```

3. **Declare Buffers**
   ```cpp
   {buffer<int, 1> Ab(A, range<1>{N});
   buffer<int, 1> Bb(B, range<1>{N});
   buffer<int, 1> Cb(C, range<1>{N});
   }
   q.submit([&] (handler& h) {
      auto R = range<1>{N};
      auto aA = Ab.get_access<access::mode::read>(h);
      auto aB = Bb.get_access<access::mode::read>(h);
      auto aC = Cb.get_access<access::mode::write>(h);
      h.parallel_for(R, [=] (id<1> i) { // A,B,C updated
          aC[i] = aA[i] + aB[i];
      });
   });
   ```
Buffer Example - Common Pattern

Create C++ Arrays

```
auto A = (int *) malloc(N * sizeof(int));
auto B = (int *) malloc(N * sizeof(int));
auto C = (int *) malloc(N * sizeof(int));
```

Initialize C++ Arrays

```
for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
}
```

Create Buffers

```
{buffer<int, 1> Ab(A, range<1>{N});
buffers<int, 1> Bb(B, range<1>{N});
buffers<int, 1> Cb(C, range<1>{N});
}
```

Create Accessors

```
q.submit([&] (handler & h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=] (id<1> i) {
        A[i] = aA[i] + aB[i];
    });
});
```

// A,B,C updated
Buffer Example – Common Pattern

Create C++ Arrays

Initialize C++ Arrays

Create Buffers

Create Accessors

Use Accessors in Kernels

auto A = (int *) malloc(N * sizeof(int));
auto B = (int *) malloc(N * sizeof(int));
auto C = (int *) malloc(N * sizeof(int));

for (int i = 0; i < N; i++) {
    A[i] = i;  B[i] = 2*i;
}

buffer<int, 1> Ab(A, range<1>{N});
buffer<int, 1> Bb(B, range<1>{N});
buffer<int, 1> Cb(C, range<1>{N});

q.submit([&] (handler& h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=] (id<1> i) {
            aC[i] = aA[i] + aB[i];
        });
});

// A,B,C updated
auto A = (int *) malloc(N * sizeof(int));
auto B = (int *) malloc(N * sizeof(int));
auto C = (int *) malloc(N * sizeof(int));

for (int i = 0; i < N; i++) {
    A[i] = i;
    B[i] = 2*i;
}

buffer<int, 1> Ab(A, range<1>{N});
buffer<int, 1> Bb(B, range<1>{N});
buffer<int, 1> Cb(C, range<1>{N});

q.submit([&](handler& h) {
    auto R = range<1>{N};
    auto aA = Ab.get_access<access::mode::read>(h);
    auto aB = Bb.get_access<access::mode::read>(h);
    auto aC = Cb.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> i) {
        aC[i] = aA[i] + aB[i];
    });
});

} // A,B,C updated
USM Example – Common Pattern

Declare USM Arrays

```c
auto A = (int *) malloc_shared(N * sizeof(int), ...);
auto B = (int *) malloc_shared(N * sizeof(int), ...);
auto C = (int *) malloc_shared(N * sizeof(int), ...);

for (int i = 0; i < N; i++) {
    A[i] = i;    B[i] = 2*i;
}

q.submit([&] (handler & h) {
    auto R = range<1>{N};
    h.parallel_for(R, [=] (id<1> ID) {
        auto i = ID[0];
        C[i] = A[i] + B[i];
    });
});
q.wait();
// A,B,C updated and ready to use
```
USM Example - Common Pattern

**Declare USM Arrays**

```cpp
auto A = (int *) malloc_shared(N * sizeof(int), ...);
auto B = (int *) malloc_shared(N * sizeof(int), ...);
auto C = (int *) malloc_shared(N * sizeof(int), ...);
```

**Initialize USM Arrays**

```cpp
for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
}
```

```cpp
q.submit([&] (handler& h) {
    auto R = range<1>{N};
    h.parallel_for(R, [=] (id<1> ID) {
        auto i = ID[0];
        C[i] = A[i] + B[i];
    });
});
q.wait();
// A,B,C updated and ready to use
```
USM Example – Common Pattern

Declare USM Arrays

Initialize USM Arrays

Read/Write USM Arrays Directly

```cpp
auto A = (int *) malloc_shared(N * sizeof(int), ...);
auto B = (int *) malloc_shared(N * sizeof(int), ...);
auto C = (int *) malloc_shared(N * sizeof(int), ...);

for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
}

q.submit([&](handler& h) {
    auto R = range<1>{N};
    h.parallel_for(R, [=] (id<1> ID) {
        auto i = ID[0];
        C[i] = A[i] + B[i];
    });
});
q.wait();
// A, B, C updated and ready to use
```

Just use the pointers!
USM Example - Common Pattern

Declare USM Arrays

Initialize USM Arrays

Read/Write USM Arrays Directly

USM Arrays Updated

```c++
auto A = (int *) malloc_shared(N * sizeof(int), ...);
auto B = (int *) malloc_shared(N * sizeof(int), ...);
auto C = (int *) malloc_shared(N * sizeof(int), ...);

for (int i = 0; i < N; i++) {
    A[i] = i; B[i] = 2*i;
}

q.submit([&] (handler& h) {
    auto R = range<1>{N};
    h.parallel_for(R, [=] (id<1> ID) {
        auto i = ID[0];
        C[i] = A[i] + B[i];
    });
});
q.wait();
// A, B, C updated and ready to use
```
Task Scheduling with USM - Options

Explicit Scheduling
- Submitting a kernel returns an Event
- Wait on Events to order tasks

DPC++ Graph Scheduling
- Build graph edges from Events

```cpp
auto E = q.submit([&](handler& h) { 
    auto R = range<1>{N};
    h.parallel_for(R, [=] (id<1> ID) {
        auto i = ID[0];
        C[i] = A[i] + B[i];
    });
});
E.wait();
```

```cpp
auto R = range<1>{N};
auto E = q.submit([&](handler& h) { 
    h.parallel_for(R, [=] (id<1> ID) {...});
});
q.submit([&](handler& h) { 
    h.depends_on(E);
    h.parallel_for(R, [=] (id<1> ID) {...});
});
```
SUB-GROUPS

Expose a grouping of work-items

- Can be mapped to vector/SIMD hardware
- Expose collective operations (e.g. shuffle, barrier, reduce)
ORDERED QUEUES

DPC++ Queues are Out-of-Order
• Allows expressing complex DAGs

Linear task chains are common
• DAGs unnecessary here and add verbosity

Simple things should be simple to express
• In-order semantics express the linear task pattern easily

// Without Ordered Queues
queue q;
auto R = range<1>{N};

auto E = q.submit([&](handler& h) {
    h.parallel_for(R, [=] (id<1> ID) {…});
});

auto F = q.submit([&](handler& h) {
    h.depends_on(E);
    h.parallel_for(R, [=] (id<1> ID) {…});
});

q.submit([&](handler& h) {
    h.depends_on(F);
    h.parallel_for(R, [=] (id<1> ID) {…});
});
DPC++ Queues are Out-of-Order

• Allows expressing complex DAGs

Linear task chains are common

• DAGs unnecessary here and add verbosity

Simple things should be simple to express

• In-order semantics express the linear task pattern easily

```cpp
// With Ordered Queues
ordered_queue q;
auto R = range<1>{N};

q.submit([&] (handler& h) {
    h.parallel_for(R, [=] (id<1> ID) {...});
});

q.submit([&] (handler& h) {
    h.parallel_for(R, [=] (id<1> ID) {...});
});

q.submit([&] (handler& h) {
    h.parallel_for(R, [=] (id<1> ID) {...});
});
```
DATA FLOW PIPES

Data with control sideband

- Fine-grained information transfer and synchronization
- Important on spatial architectures (FPGA)
The SYCL 1.2.1 standard requires all kernels to have a unique name:

- Functor class type
- Template typename for Lambdas

DPC++ removes this requirement for Lambdas:

- Must use DPC++ compiler for both host and device code
- Enabled via compiler switch
  - -fsycl-unnamed-lambda

```cpp
q.submit([&] (handler& h) {
    auto R = range<1>{N};
    h.parallel_for(R, [=](id<1> ID) {
        auto i = ID[0];
        C[i] = A[i] + B[i];
    });
});
```
DPC++ is a standards-based, cross-architecture language to deliver uncompromised productivity and performance across CPUs and accelerators

• Extends the SYCL 1.2.1 standard with new features

Kernel-based model, with task graph

New features being developed through a community project
• https://github.com/intel/llvm
• Feel free to open an Issue or submit a PR!
Start with **oneAPI today** for an accelerated xPU future

Get the **oneAPI specification** at
oneAPI.com

---

**Use Intel’s oneAPI Beta**

Test code and workloads across a range of Intel® data-centric architectures at
**Intel® DevCloud for oneAPI**
software.intel.com/devcloud/oneAPI

---

Learn more and download the **beta toolkits** at
software.intel.com/oneapi
oneAPI available now on Intel DevCloud

A development sandbox to develop, test and run your workloads across a range of Intel CPUs, GPUs, and FPGAs using Intel's oneAPI beta software

software.intel.com/en-us/devcloud/oneapi

Get up and running in seconds!
NOTICES & DISCLAIMERS

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Copyright © 2019, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and OpenVINO are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries. Khronos® is a registered trademark and SYCL is a trademark of the Khronos Group, Inc.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804