// Compute sum of length-N vectors: C = A + B
void vecAdd (float* a, float* b, float* c, int N) {
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    a = new float[N];
    // ... allocate other arrays, fill with data

    vecAdd (a, b, c, N);
}
Hello World: Vector Addition

// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a, sizeof(float) * N);
    // ... allocate other arrays, fill with data

    // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
Cuda Software Environment

- nvcc compiler works much like icc or gcc: compiles C++ source code, generates binary executable
- Nvidia Cuda OS driver manages low-level interaction with device, provides API for C++ programs
- Nvidia Cuda SDK has many code samples demonstrating various Cuda functionalities
- Library support is continuously growing:
  - CUBLAS for basic linear algebra
  - CUFFT for Fourier Transforms
  - CULapack (3rd party proprietary) linear solvers, eigensolvers, ...
- OS-Portable: Linux, Windows, Mac OS
- A lot of momentum in Industrial adoption of Cuda!

http://developer.nvidia.com/object/cuda_3_1_downloads.html
Agenda

• A Shameless self-promotion
• Introduction to GPGPUs and Cuda Programming Model
• The Cuda Thread Hierarchy
• The Cuda Memory Hierarchy
• Mapping Cuda to Nvidia GPUs
• As much of the OpenCL information as I can get through
Nvidia Cuda GPU Architecture

• I'll discuss some details of Nvidia's GPU architecture simultaneously with discussing the Cuda Programming Model
  – The Cuda Programming Model is a set of data-parallel extensions to C, amenable to implementation on GPUs, CPUs, FPGAs, ...

• Cuda GPUs are a collection of “Streaming Multiprocessors”
  – Each SM is analogous to a core of a Multi-Core CPU

• Each SM is a collection of SIMD execution pipelines (Scalar Processors) that share control logic, register file, and L1 Cache
Cuda Thread Hierarchy

- Parallelism in the Cuda Programming Model is expressed as a 4-level Hierarchy:
  - A **Stream** is a list of **Grids** that execute in-order. Fermi GPUs execute multiple Streams in parallel.
  - A **Grid** is a set of up to $2^{32}$ **Thread Blocks** executing the same kernel.
  - A **Thread Block** is a set of up to 1024 [512 pre-Fermi] **Cuda Threads**.
  - Each **Cuda Thread** is an independent, lightweight, scalar execution context.
    - Groups of 32 threads form **Warps** that execute in lockstep SIMD.
What is a Cuda Thread?

• Logically, each Cuda Thread is its own very lightweight independent MIMD execution context
  – Has its own control flow and PC, register file, call stack, ...
  – Can access any GPU global memory address at any time
  – Identifiable uniquely within a grid by the five integers: threadIdx.{x,y,z}, blockIdx.{x,y}

• Very fine granularity: do not expect any single thread to do a substantial fraction of an expensive computation
  – At full occupancy, each Thread has 21 32-bit registers
  – ... 1,536 Threads share a 64 KB L1 Cache / __shared__ mem
  – GPU has no operand bypassing networks: functional unit latencies must be hidden by multithreading or ILP (e.g. from loop unrolling)
Cuda Memory Hierarchy

- Thread blocks in all Grids share access to a large pool of “Global” memory, separate from the Host CPU’s memory.
  - Global memory holds the application’s persistent state, while the thread-local and block-local memories are temporary
  - Global memory is much more expensive than on-chip memories: $O(100)x$ latency, $O(1/50)x$ (aggregate) bandwidth
- On Fermi, Global Memory is cached in a 768KB shared L2
Cuda Memory Hierarchy

• There are other read-only components of the Memory Hierarchy that exist due to the Graphics heritage of Cuda

• The 64 KB Cuda **Constant Memory** resides in the same DRAM as global memory, but is accessed via special read-only 8 KB per-SM caches

• The Cuda **Texture Memory** also resides in DRAM and is accessed via small per-SM read-only caches, but also includes interpolation hardware
  – This hardware is crucial for graphics performance, but only occasionally is useful for general-purpose workloads

• The behaviors of these caches are highly optimized for their roles in graphics workloads.
Cuda Memory Hierarchy

- Each Cuda device in the system has its own Global memory, separate from the Host CPU memory
  - Allocated via cudaMalloc()/cudaFree() and friends
- Host ↔ Device memory transfers are via cudaMemcpy() over PCI-E, and are extremely expensive
  - Microsecond latency, ~GB/s bandwidth
- Multiple Devices managed via multiple CPU threads
Thread-Block Synchronization

• Intra-block barrier instruction `__syncthreads()` for synchronizing accesses to `__shared__` and global memory
  – To guarantee correctness, must `__syncthreads()` before reading values written by other threads
  – All threads in a block must execute the same `__syncthreads()`, or the GPU will hang (not just the same number of barriers !)

• Additional intrinsics worth mentioning here:
  – `int __syncthreads_count(int)`, `int __syncthreads_and(int)`, `int __syncthreads_or(int)`

```c
extern __shared__ float T[];
__device__ void
transpose (float* a, int lda){
    int i = threadIdx.x, j = threadIdx.y;
    T[i + lda*j] = a[i + lda*j];
    __syncthreads();
    a[i + lda*j] = T[j + lda*i];
}
```
Using per-block shared memory

- The per-block shared memory / L1 cache is a crucial resource: without it, the performance of most Cuda programs would be hopelessly DRAM-bound
- Block-shared variables can be declared statically:
  ```
  __shared__ int begin, end;
  ```
- Software-managed scratchpad is allocated statically:
  ```
  __shared__ int scratch[128];
  scratch[threadIdx.x] = ... ;
  ```
- ... or dynamically:
  ```
  extern __shared__ int scratch[];
  ```
  ```
  kernel_call <<< grid_dim, block_dim, scratch_size >>> ( ... );
  ```
- Most intra-block communication is via shared scratchpad:
  ```
  scratch[threadIdx.x] = ...;
  __syncthreads();
  int left = scratch[threadIdx.x - 1];
  ```
Using per-block shared memory

• Each SM has 64 KB of private memory, divided 16KB/48KB (or 48KB/16KB) into software-managed scratchpad and hardware-managed, non-coherent cache
  – Pre-Fermi, the SM memory is only 16 KB, and is usable only as software-managed scratchpad

• Unless data will be shared between Threads in a block, it should reside in registers
  – On Fermi, the 128 KB Register file is twice as large, and accessible at higher bandwidth and lower latency
  – Pre-Fermi, register file is 64 KB and equally fast as scratchpad
Shared Memory Bank Conflicts

- Shared memory is **banked**: it consists of 32 (16, pre-Fermi) independently addressable 4-byte wide memories
  - Addresses interleave: `float *p` points to a float in bank \( k \), \( p+1 \) points to a float in bank \( (k+1) \mod 32 \)
- Each bank can satisfy a single 4-byte access per cycle.
  - A **bank conflict** occurs when two threads (in the same warp) try to access the same bank in a given cycle.
  - The GPU hardware will execute the two accesses serially, and the warp's instruction will take an extra cycle to execute.
- Bank conflicts are a second-order performance effect: even serialized accesses to on-chip shared memory is faster than accesses to off-chip DRAM
Shared Memory Bank Conflicts

- Figure G-2 from Cuda C Programming Guide 3.1
- Unit-Stride access is conflict-free
- Stride-2 access: thread $n$ conflicts with thread $16+n$
- Stride-3 access is conflict-free
Shared Memory Bank Conflicts

- Three more cases of conflict-free access
  - Figure G-3 from Cuda C Programming Guide 3.1

- Permutations within a 32-float block are OK

- Multiple threads reading the same memory address

- All threads reading the same memory address is a broadcast
Atomic Memory Operations

- Cuda provides a set of instructions which execute atomically with respect to each other
  - Allow non-read-only access to variables shared between threads in shared or global memory
  - Substantially more expensive than standard load/stores
  - With voluntary consistency, can implement e.g. spin locks!

```c
int atomicAdd (int*, int), float atomicAdd (float*, float), ...
...
int atomicMin (int*, int),
...
int atomicExch (int*, int), float atomicExch (float*, float), ...
int atomicCAS (int*, int compare, int val), ...
```
Voluntary Memory Consistency

• By default, you cannot assume memory accesses are occur in the same order specified by the program
  – Although a thread's own accesses appear to that thread to occur in program order

• To enforce ordering, use memory fence instructions
  – __threadfence_block(): make all previous memory accesses visible to all other threads within the thread block
  – __threadfence(): make previous global memory accesses visible to all other threads on the device

• Frequently must also use the volatile type qualifier
  – Has same behavior as CPU C/C++: the compiler is forbidden from register-promoting values in volatile memory
  – Ensures that pointer dereferences produce load/store instructions
  – Declared as volatile float *p; *p must produce a memory ref.
Introduction to OpenCL

Tim Mattson
Microprocessor and Programming Research Lab
Intel Corp.
Agenda

• Ugly programming models and why they rule
  • The origin of OpenCL
  • A high level view of OpenCL
  • OpenCL and the CPU
  • An OpenCL “deep dive”
Heterogeneous computing

• A modern platform has:
  - Multi-core CPU(s)
  - A GPU
  - DSP processors
  - ... other?

• The goal should NOT be to “off-load” the CPU. We need to make the best use of all the available resources from within a single program:
  - One program that runs well (i.e. reasonably close to “hand-tuned” performance) on a heterogeneous mixture of processors.

OpenCL: 21
Heterogeneous many core processors

The mass market hardware landscape has never been so chaotic … and its only going to get worse.

Intel Dual Core CPU

IBM Cell

Intel 80 core research chip

NVIDIA 8800

3rd party names are the property of their owners.
The many-core challenge

• We have arrived at many-core solutions *not* because of the success of our parallel software but because of our *failure* to keep increasing CPU frequency.

• Result: a fundamental and dangerous mismatch
  - Parallel hardware is ubiquitous.
  - Parallel software is rare

Our challenge ... make parallel software as routine as our parallel hardware.
Patterns and Frameworks

• In the long run, we will provide high level frameworks/scripting-languages that will meet the needs of the domain-expert, application programmers (we hope).
  - Design patterns will guide us to the right framework designs.

• But even in a frameworks world, you need to support the framework programmers
  - (also known as efficiency programmers, technology programmers, socially mal-adjusted performance hackers, etc)

• How do we support these low-level “performance obsessed” programmers?
### Solution: Find A Good parallel programming model, right?

<table>
<thead>
<tr>
<th>Models from the golden age of parallel programming</th>
</tr>
</thead>
</table>
The only thing sillier than creating too many models is using too many

Programming models I’ve worked with.

8/19/10  Tim Mattson  OpenCL: 26
There is nothing new under the sun

- Message passing models:
  - MPI    PVM
- Data Parallel programming models
  - C*     HPF    NESL    CMFortran
- Virtual Shared Memory models
  - Linda   GA
- Functional Languages
  - Haskell SISAL
- Formal compositional models
  - CC++    PCN
- Shared address space ... threads
  - OpenMP  Cilk
- Parallel object Oriented programming
  - Mentat  CHARM++  POOMA  TBB

Parallel programming ... “been there, done that”
Will we be wise enough to learn from the past?
Lesson 1: computer scientists are easily seduced by beauty

• A beautiful programming model:
  - Safe: it’s hard to do bad things
  - Expressive: focus on the intent of the algorithm.
  - Abstract: Hides hardware details
  - Novel: New ideas and fresh perspectives

To the computer scientist … There is no problem that can’t be solved by adding another layer of abstraction.
The history of parallel programming can be viewed as computer scientists chasing after an elusive ideal of beauty.
Lesson 2: Software vendors (not academics and not hardware vendors) choose the winning programming models

- What software developers need:
  - Portability: recompile to run on every platform the market demands
  - Stability: program life times measured in decades.
  - Predictability: the ability to build code that adapts to hardware details for predictable performance.

Industry standards with minimal HW constraints

Established prog. Envs. from long term, trusted sources

HW details exposed so SW can adapt
Ugly programming models win!

- Software developers only weakly care about beauty in a programming model ... pragmatism wins.
- History supports ugly programming models ... with all the elegant abstractions for parallelism that have been created, what is actually used:
  - MPI
  - Explicit thread libraries
  - Compiler directives

OpenCL is truly ugly ... and to support our framework developers facing heterogenous many core platforms, its exactly what we need!
... just look at all the built-in functions we had to define to make this thing work?
Agenda

• Ugly programming models and why they rule
• The origin of OpenCL
• A high level view of OpenCL
• OpenCL and the CPU
• An OpenCL “deep dive”
OpenCL – Open Computing Language
Open standard for portable programming of heterogeneous platforms (CPUs, GPUs, and other processors)
Consider the historical precedent with OpenMP ...

**SGI**
- Merged, needed commonality across products

**Cray**
- ISV - needed larger market

**KAI**
- ISV - needed larger market
- was tired of recoding for SMPs. Forced vendors to standardize.

**ASCI**
- Wrote a rough draft straw man SMP API

Other vendors invited to join

1997

Third party names are the property of their owners.
OpenCL: Can history repeat itself?

As ASCI did for OpenMP, Apple is doing for GPU/CPU with OpenCL

AMD

Merged, needed commonality across products

ATI

Nvidia

GPU vendor - wants to steal mkt share from CPU

Intel

CPU vendor - wants to steal mkt share from GPU

Apple

was tired of recoding for many core, GPUs. Pushed vendors to standardize.

Wrote a rough draft straw man API

Khronos Compute group formed

Dec 2008

Third party names are the property of their owners.
OpenCL Working Group

- Designed with real users (Apple + ISVs) to solve their problems.
- Used Khronos to make it an industry standard.
OpenCL Timeline

- Six months from proposal to released specification
- Commercial support:
  - Apple’s Mac OS X Snow Leopard (9’2009) will include OpenCL.
  - Nvidia OpenCL beta release on CUDA.
  - AMD released a CPU OpenCL SIGGRAPH’09
  - Intel actively promotes OpenCL, but we have not announced our product strategy for OpenCL yet.
OpenCL 1.0 Embedded Profile

• Enables OpenCL on mobile and embedded silicon
  – Relaxes some data type and precision requirements
  – Avoids the need for a separate “ES” specification

• Khronos APIs provide computing support for imaging & graphics
  – Enabling advanced applications in, e.g., Augmented Reality

• OpenCL will enable parallel computing in new markets
  – Mobile phones, cars, avionics

Source: Kari Pulli, Nokia

A camera phone with GPS processes images to recognize buildings and landmarks and provides relevant data from internet
Agenda

• Ugly programming models and why they rule
• The origin of OpenCL
• A high level view of OpenCL
• OpenCL and the CPU
• An OpenCL “deep dive”
OpenCL: high level view

• OpenCL applications:
  - A host program running on the PC
  - One or more *Kernels* that are queued up to run on CPUs, GPUs, and “other processors”.

• OpenCL is understood in terms of these models
  - Platform model
  - Execution model
  - Memory model
  - Programming model
OpenCL Platform model

The basic platform is a host and one or more compute devices.
Execution Model

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue.

Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract **Index Space**.

A \((G_y \times G_x)\) index space.

Work items execute together as a **work-group**.
OpenCL Memory model

- Implements a relaxed consistency, shared memory model

- Global memory: visible to host and compute devices
- Private memory: Local to each work item
- Local memory: Shared within a work group

Compute Device

Compute unit 1

Private memory 1

PE 1

Local memory 1

... 

Compute unit N

Private memory 1

PE 1

Local memory N

Global/Constant Memory Data Cache

Global Memory

Constant Memory

Compute Device Memory
OpenCL programming model

• Data Parallel, SPMD
  - Work-items in a work-group run the same program
  - Update data structures in parallel using the work-item ID to select data and guide execution.

• Task Parallel
  - One work-item per work group ... for coarse grained task-level parallelism.
  - Native function interface: trap-door to run arbitrary code from an OpenCL command-queue.
• Derived from ISO C99
  – No standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
• Additions to the language for parallelism
  – Work-items and workgroups
  – Vector types
  – Synchronization
• Address space qualifiers
• Optimized image access
• Built-in functions

Acknowledgement: Aaftab Munshi of Apple
OpenCL C: Data Types

- **Scalar data types**
  - char, uchar, short, ushort, int, uint, long, ulong
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)

- **Image types**
  - image2d_t, image3d_t, sampler_t

- **Vector data types**

Acknowledgement: Aaftab Munshi of Apple
OpenCL C: Vector Types

- Portable
- Vector length of 2, 4, 8, and 16
- char2, ushort4, int8, float16, ...
- Endian safe
- Aligned at vector length
- Vector operations and built-in functions

Acknowledgement: Aaftab Munshi of Apple
Vector Operations

- **Vector literal**

  ```c
  int4 vi0 = (int4) -7;
  int4 vil = (int4)(0, 1, 2, 3);
  ```

- **Vector components**

  ```c
  vi0.lo = vil.hi;
  ```

- **Vector ops**

  ```c
  vi0 += vil;
  vi0 = abs(vi0);
  ```

Acknowledgement: Aaftab Munshi of Apple
OpenCL Software Stack

- **Platform Layer**:
  - query and select compute devices
  - create contexts and command-queues

- **Runtime**
  - Coordinate between host and Compute devices
  - resource management
  - execute kernels

- **Compiler**
  - Implements kernel code on Target Device
  - ISO C99 subset + a few language additions
  - Builds executables online or offline
Example: vector addition

• The “hello world” program of data parallel programming is a program to add two vectors

\[ C[i] = A[i] + B[i] \quad \text{for } i=1 \text{ to } N \]

• For the OpenCl solution, there are two parts
  - Kernel code
  - Host code
Vector Addition - Kernel

__kernel void vec_add (__global const float *a,
                      __global const float *b,
                      __global      float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
Vector Addition - Host Program

// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
  -CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
cGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
  NULL, &cb);
devices = malloc(cb);
cGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
  devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
  0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
  CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
  NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
  CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
  NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
  sizeof(cl_float)*n, NULL,
  NULL);

// create the program
program = clCreateProgramWithSource(context, 1,
  &program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
  NULL);

// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
  sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1],
  sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
  sizeof(cl_mem));

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
  NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE,
  0, n*sizeof(cl_float), dst, 0, NULL, NULL);

The host program is ugly ... but its not too hard to understand (details with readable font in back-up slides)
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
    CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
    NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
    devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
    0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
    NULL);;
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
    NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    sizeof(cl_float)*n, NULL,
    NULL);

// create the program
program = clCreateProgramWithSource(context, 1,
    &program_source, NULL, NULL);

Build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// set the args values
err = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
    sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
    sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
    sizeof(cl_mem));

Create and setup kernel

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
    NULL, global_work_size, NULL, 0, NULL, NULL);

Execute the kernel

// read output array
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE,
    0, n*sizeof(cl_float), dst, 0, NULL, NULL);

Read results on the host

OpenCL: 53
Agenda

• Ugly programming models and why they rule
• The origin of OpenCL
• A high level view of OpenCL
• OpenCL and the CPU
• An OpenCL “deep dive”
OpenCL’s Two Styles of Data-Parallelism

• Explicit SIMD data parallelism:
  - The kernel defines one stream of instructions
  - Parallelism from using wide vector types
  - Size vector types to match native HW width
  - Combine with task parallelism to exploit multiple cores.

• Implicit SIMD data parallelism (i.e. shader-style):
  - Write the kernel as a “scalar program”
  - Use vector data types sized naturally to the algorithm
  - Kernel automatically mapped to SIMD-compute-resources and cores by the compiler/runtime/hardware.

Both approaches are viable CPU options
Data-Parallelism: options on IA processors

- Explicit SIMD data parallelism
  - Programmer chooses vector data type (width)
  - Compiler hints using attributes
    » vec_type_hint(tyben)

- Implicit SIMD Data parallel
  - Map onto CPUs, GPUs, Larrabee, ...
    » SSE/AVX/LRBni: 4/8/16 workitems in parallel

- Hybrid use of the two methods
  » AVX: can run two 4-wide workitems in parallel
  » LRBni: can run four 4-wide workitems in parallel
Explicit SIMD data parallelism

- OpenCL as a portable interface to vector instruction sets.
  - Block loops and pack data into vector types (float4, ushort16, etc).
  - Replace scalar ops in loops with blocked loops and vector ops.
  - Unroll loops, optimize indexing to match machine vector width

```c
float a[N], b[N], c[N];
for (i=0; i<N; i++)
  c[i] = a[i]*b[i];
<<< the above becomes >>>>
float4 a[N/4], b[N/4], c[N/4];
for (i=0; i<N/4; i++)
  c[i] = a[i]*b[i];
```

Explicit SIMD data parallelism means you tune your code to the vector width and other properties of the compute device.
Video Processing Case Study

• 2 algorithms from the Video Processing domain
  - Color Enhancement
    » Enhance the saturation (color strength) of individual colors
      • Red, Green, Blue, Yellow, Cyan and Magenta
  - Contrast Enhancement
    » Improve extreme dark and bright images

• Video Frames
  - Processed in YUV 4:2:0 planar color space
  - 10 bits per color component
    » Contained in ushort (unsigned short)
  - Fixed point arithmetic
  - Structure of arrays (SOA)
**Explicit SIMD data parallelism: Case Study**

- Video contrast/color optimization kernel on a dual core CPU.

- **Hand-tuned SSE + Multithreading**
- **Unroll loops**
- **Optimize vector indexing**
- **Vectorize (block loops, pack into ushort8 and ushort16)**
- **1 work-item per core + loops**

**Successive improvement**

- 5% improvement
- 23% improvement
- 186% improvement
- 40% improvement

**% peak performance**

- 20%
- 5%
- 100%

**Good news:** OpenCL code 95% of hand-tuned SSE/MT perf.

**Bad news:** New platform, redo all those optimizations.

---

*Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.*

---

3 Ghz dual core CPU
pre-release version of OpenCL
Source: Intel Corp.
Towards “Portable” Performance

• The following C code is an example of a Bilateral 1D filter:

```c
void P4_Bilateral9 (int start, int end, float v)
{
    int i, j, k;
    float w[4], a[4], p[4];
    float inv_of_2v = -0.5 / v;
    for (i = start; i < end; i++) {
        float wt[4] = { 1.0f, 1.0f, 1.0f, 1.0f};
        for (k = 0; k < 4; k++)
            a[k] = image[i][k];
        for (j = 1; j <= 4; j++) {
            for (k = 0; k < 4; k++)
                p[k] = image[i - j*SIZE][k] - image[i][k];
            for (k = 0; k < 4; k++)
                w[k] = exp (p[k] * p[k] * inv_of_2v);
            for (k = 0; k < 4; k++)
                wt[k] += w[k];
            for (k = 0; k < 4; k++)
                a[k] += w[k] * image[i - j*SIZE][k];
        }
        for (j = 1; j <= 4; j++) {
            for (k = 0; k < 4; k++)
                p[k] = image[i + j*SIZE][k] - image[i][k];
            for (k = 0; k < 4; k++)
                w[k] = exp (p[k] * p[k] * inv_of_2v);
            for (k = 0; k < 4; k++)
                wt[k] += w[k];
            for (k = 0; k < 4; k++)
                a[k] += w[k] * image[i + j*SIZE][k];
        }
        for (k = 0; k < 4; k++)
            image2[i][k] = a[k] / wt[k];
    }
}
```

• Reminder: Bilateral filter is an edge preserving image processing algorithm.

• See more information here: http://scien.stanford.edu/class/psych221/projects/06/imagescaling/bilati.html
The following C code is an example of a Bilateral 1D filter:

Reminder: Bilateral filter is an edge preserving image processing algorithm.

See more information here: http://scien.stanford.edu/class/psych221/projects/06/imagescaling/bilati.html

```c
void P4_Bilateral9 (int start, int end, float v) {
    int i, j, k;
    float w[4], a[4], p[4];
    float inv_of_2v = -0.5 / v;
    for (i = start; i < end; i++) {
        float wt[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
        for (k = 0; k < 4; k++)
            a[k] = image[i][k];
        for (j = 1; j <= 4; j++) {
            for (k = 0; k < 4; k++)
                p[k] = image[i - j*SIZE][k] - image[i][k];
            for (k = 0; k < 4; k++)
                w[k] = exp (p[k] * p[k] * inv_of_2v);
            for (k = 0; k < 4; k++) {
                wt[k] += w[k];
                a[k] += w[k] * image[i - j*SIZE][k];
            }
        }
        for (j = 1; j <= 4; j++) {
            for (k = 0; k < 4; k++)
                p[k] = image[i + j*SIZE][k] - image[i][k];
            for (k = 0; k < 4; k++)
                w[k] = exp (p[k] * p[k] * inv_of_2v);
            for (k = 0; k < 4; k++) {
                wt[k] += w[k];
                a[k] += w[k] * image[i + j*SIZE][k];
            }
        }
        for (k = 0; k < 4; k++)
            image2[i][k] = a[k] / wt[k];
    }
}
```

Source: Intel Corp.

Towards “Portable” Performance
**“Implicit SIMD” data parallel code**

- “outer” loop replaced by work-items running over an NDRange index set.

- NDRange $4 \times$ image size ... since each workitem does a color for each pixel.

- Leave it to the compiler to map work-items onto lanes of the vector units ...

```
__kernel void P4_Bilateral9 (__global float* inImage, __global float* outImage, float v)
{
    const size_t myID   = get_global_id(0);
    const float inv_of_2v = -0.5f / v;
    const size_t myRow   = myID / IMAGE_WIDTH;
    size_t maxDistance = min(DISTANCE, myRow);
    maxDistance = min(maxDistance, IMAGE_HEIGHT - myRow);
    float currentPixel, neighborPixel, newPixel;
    float diff;
    float accumulatedWeights, currentWeights;
    newPixel = currentPixel = inImage[myID];
    accumulatedWeights = 1.0f;
    for (size_t dist = 1; dist <= maxDistance; ++dist)
    {
        neighborPixel            = inImage[myID + dist*IMAGE_WIDTH];
        diff                           = neighborPixel - currentPixel;
        currentWeights         = exp(diff * diff * inv_of_2v);
        accumulatedWeights += currentWeights;
        newPixel                   += neighborPixel * currentWeights;
        neighborPixel              = inImage[myID - dist*IMAGE_WIDTH];
        diff                             = neighborPixel - currentPixel;
        currentWeights           = exp(diff * diff * inv_of_2v);
        accumulatedWeights += currentWeights;
        newPixel                   += neighborPixel * currentWeights;
    }
    outImage[myID] = newPixel / accumulatedWeights;
}
```

Source: Intel Corp.
“Implicit SIMD” data parallel code

```c
__kernel void p4_bilateral9(__global float* inImage, __global float* outImage, float v)
{
    const size_t myID     = get_global_id(0);
    const float inv_of_2v = -0.5f / v;
    const size_t myRow    = myID / IMAGE_WIDTH;
    size_t maxDistance = min(DISTANCE, myRow);
    maxDistance = min(maxDistance, IMAGE_HEIGHT - myRow);
    float currentPixel, neighborPixel, newPixel;
    float diff;
    float accumulatedWeights, currentWeights;
    newPixel = currentPixel = inImage[myID];
    accumulatedWeights = 1.0f;
    for (size_t dist = 1; dist <= maxDistance; ++dist)
    {
        neighborPixel = inImage[myID + dist*IMAGE_WIDTH];
        diff = neighborPixel - currentPixel;
        currentWeights = exp(diff * diff * inv_of_2v);
        accumulatedWeights += currentWeights;
        newPixel += neighborPixel * currentWeights;
        neighborPixel = inImage[myID - dist*IMAGE_WIDTH];
        diff = neighborPixel - currentPixel;
        currentWeights = exp(diff * diff * inv_of_2v);
        accumulatedWeights += currentWeights;
        newPixel += neighborPixel * currentWeights;
    }
    outImage[myID] = newPixel / accumulatedWeights;
}
```

Source: Intel Corp.
Portable Performance in OpenCL

- Implicit SIMD code ... where the framework maps work-items onto the “lanes of the vector unit” ... creates the opportunity for portable code that performs well on full range of OpenCL compute devices.

- Requires mature OpenCL technology that “knows” how to do this:
  - ... But it is important to note .... we know this approach works since its based on the way shader compilers work today.
Task Parallelism Overview

- Think of a task as an asynchronous function call
  - “Do X at some point in the future”
  - Optionally “... after Y is done”
  - Light weight, often in user space

- Strengths
  - Copes well with heterogeneous workloads
  - Doesn’t require 1000’s of strands
  - Scales well with core count

- Limitations
  - No automatic support for latency hiding
  - Must explicitly write SIMD code

A natural fit to multi-core CPUs
Task Parallelism in OpenCL

• **clEnqueueTask**
  - Imagine “sea of different tasks” executing concurrently
  - A task “owns the core” (i.e., a workgroup size of 1)

• **Use tasks when algorithm...**
  - Benefits from large amount of local/private memory
  - Has predictable global memory accesses
  - Can be programmed using explicit vector style
  - “Just doesn’t have 1000’s of identical things to do”

• **Use data-parallel kernels when algorithm...**
  - Does not benefit from large amounts of local/private memory
  - Has unpredictable global memory accesses
  - Needs to apply same operation across large number of data elements
Future Parallel Programming

• Real world applications contain data parallel parts as well as serial/sequential parts

• OpenCL addresses these Apps need by supporting Data Parallel & Task Parallel

• “Braided Parallelism” – composing Data Parallel & Task Parallel constructs in a single algorithm

• CPUs are ideal for Braided Parallelism
Future parallel programming: Larrabee

- Cores communicate on a wide ring bus
  - Fast access to memory and fixed function blocks
  - Fast access for cache coherency
- L2 cache is partitioned among the cores
  - Provides high aggregate bandwidth
  - Allows data replication & sharing

OpenCL: 68
Processor Core Block Diagram

• Separate scalar and vector units with separate registers
• Vector unit: 16 32-bit ops/clock
• In-order instruction execution
• Short execution pipelines
• Fast access from L1 cache
• Direct connection to each core’s subset of the L2 cache
• Prefetch instructions load L1 and L2 caches
Key Differences from Typical GPUs

- Each Larrabee core is a complete Intel processor
  - Context switching & pre-emptive multi-tasking
  - Virtual memory and page swapping, even in texture logic
  - Fully coherent caches at all levels of the hierarchy

- Efficient inter-block communication
  - Ring bus for full inter-processor communication
  - Low latency high bandwidth L1 and L2 caches
  - Fast synchronization between cores and caches

Larrabee is perfect for the braided parallelism in future applications
Conclusion

• OpenCL defines a platform-API/framework for heterogeneous computing ... not just GPGPU or CPU-offload programming.

• OpenCL has the potential to deliver portably performant code; but only if its used correctly:
  - Implicit SIMD data parallel code has the best chance of mapping onto a diverse range of hardware ... once OpenCL implementation quality catches up with mature shader languages.

• The future is clear:
  - Parallelism mixing task parallel and data parallel code in a single program ... balancing the load among \textsc{All Of} the platform’s available resources.
  - OpenCL can handle this ... and emerging platforms (e.g. Larrabee) will increasingly emphasize this model.
References

- s09.idav.ucdavis.edu for slides from a Siggraph2009 course titled “Beyond Programmable Shading”


Agenda

- Ugly programming models and why they rule
- The origin of OpenCL
- A high level view of OpenCL
- OpenCL and the CPU
- An OpenCL “deep dive”
Basic OpenCL Program Structure

• Host program
  - Query compute devices
  - Create contexts
  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources

• Kernels
  - C code with some restrictions and extensions
Example: Vector Addition

• Compute \( c = a + b \)
  - \( a, b, \) and \( c \) are vectors of length \( N \)

• Basic OpenCL concepts
  - Simple kernel code
  - Basic context management
  - Memory allocation
  - Kernel invocation
Platform Layer: Basic discovery

- Platform layer allows applications to query for platform specific features
- Querying platform info Querying devices
  - `clGetDeviceIDs()`
    - Find out what compute devices are on the system
    - Device types include CPUs, GPUs, or Accelerators
  - `clGetDeviceInfo()`
    - Queries the capabilities of the discovered compute devices such as:
      - Number of compute cores
      - Maximum work-item and work-group size
      - Sizes of the different memory spaces
      - Maximum memory object size
Platform Layer: Contexts

- Creating contexts
  - Contexts are used by the OpenCL runtime to manage objects and execute kernels on one or more devices
  - Contexts are associated to one or more devices
    » Multiple contexts could be associated to the same device
  - `clCreateContext()` and `clCreateContextFromType()` returns a handle to the created contexts
Platform layer: Command-Queues

- Command-queues store a set of operations to perform
- Command-queues are associated to a context
- Multiple command-queues can be created to handle independent commands that don’t require synchronization
- Execution of the command-queue is guaranteed to be completed at sync points
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0, // (must be 0)
    CL_DEVICE_TYPE_GPU,
    NULL, // error callback
    NULL, // user data
    NULL); // error code

// get the list of GPU devices associated with context
size_t cb;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id *devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cl_cmd_queue cmd_queue = clCreateCommandQueue(context,
    devices[0], 0, // default options
    NULL); // error code
Memory Objects

• **Buffer objects**
  - One-dimensional collection of objects (like C arrays)
  - Valid elements include scalar and vector types as well as user defined structures
  - Buffer objects can be accessed via pointers in the kernel

• **Image objects**
  - Two- or three-dimensional texture, frame-buffer, or images
  - Must be addressed through built-in functions

• **Sampler objects**
  - Describes how to sample an image in the kernel
    » Addressing modes
    » Filtering modes
Creating Memory Objects

- `clCreateBuffer()`, `clCreateImage2D()`, and `clCreateImage3D()`
- Memory objects are created with an associated context
- Memory can be created as read only, write only, or read-write
- Where objects are created in the platform memory space can be controlled
  - Device memory
  - Device memory with data copied from a host pointer
  - Host memory
  - Host memory associated with a pointer
    » Memory at that pointer is guaranteed to be valid at synchronization points
Manipulating Object Data

- Object data can be copied to host memory, from host memory, or to other objects.
- Memory commands are enqueued in the command buffer and processed when the command is executed:
  - `clEnqueueReadBuffer()`, `clEnqueueReadImage()`
  - `clEnqueueWriteBuffer()`, `clEnqueueWriteImage()`
  - `clEnqueueCopyBuffer()`, `clEnqueueCopyImage()`
- Data can be copied between Image and Buffer objects:
  - `clEnqueueCopyImageToBuffer()`
  - `clEnqueueCopyBufferToImage()`
- Regions of the object data can be accessed by mapping into the host address space:
  - `clEnqueueMapBuffer()`, `clEnqueueMapImage()`
  - `clEnqueueUnmapMemObject()`
VecAdd: Create Memory Objects

```c
cl_mem memobjs[3];

// allocate input buffer memory objects
memobjs[0] = clCreateBuffer(context,
       CL_MEM_READ_ONLY | // flags
       CL_MEM_COPY_HOST_PTR,
       sizeof(cl_float)*n, // size
       srcA,                // host pointer
       NULL);               // error code

memobjs[1] = clCreateBuffer(context,
       CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
       sizeof(cl_float)*n, srcB, NULL);

// allocate input buffer memory object
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
       sizeof(cl_float)*n, NULL, NULL);
```
Program Objects

- Program objects encapsulate:
  - An associated context
  - Program source or binary
  - List of targeted devices, build options
  - Number of attached kernel objects

- Build process
  1. Create program object
     » clCreateProgramWithSource()
     » clCreateProgramWithBinary()
  2. Build program executable
     » Compile and link from source or binary for all devices or specific devices in the associated context
     » clBuildProgram()
     » Build options
        • Preprocessor, float point behavior, optimizations, etc
Kernel Objects

- Kernel objects encapsulate
  - Specific kernel functions declared in a program
  - Argument values used for kernel execution

- Creating kernel objects
  - `clCreateKernel()` - creates a kernel object for a single function in a program

- Setting arguments
  - `clSetKernelArg(<kernel>, <argument index>)`
  - Each argument data must be set for the kernel function
  - Argument values copied and stored in the kernel object

- Kernel vs. program objects
  - Kernels are related to program execution
  - Programs are related to program source
VecAdd: Program and Kernel

// create the program
cl_program program = clCreateProgramWithSource(
    context,
    1, // string count
    &program_source, // program strings
    NULL, // string lengths
    NULL); // error code

// build the program
cl_int err = clBuildProgram(program,
    0, // num devices in device list
    NULL, // device list
    NULL, // options
    NULL, // notifier callback function ptr
    NULL); // user data

// create the kernel
cl_kernel kernel = clCreateKernel(program, "vec_add", NULL);
// set "a" vector argument
err = clSetKernelArg(kernel,
  0,                   // argument index
  (void *)&memobjs[0], // argument data
  sizeof(cl_mem)));    // argument data size

// set "b" vector argument
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
  sizeof(cl_mem));

// set "c" vector argument
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
  sizeof(cl_mem));
Kernel Execution

- A command to execute a kernel must be enqueued to the command-queue
- `clEnqueueNDRangeKernel()`
  - Data-parallel execution model
  - Describes the *index space* for kernel execution
  - Requires information on NDRange dimensions and work-group size
- `clEnqueueTask()`
  - Task-parallel execution model (multiple queued tasks)
  - Kernel is executed on a single work-item
- `clEnqueueNativeKernel()`
  - Task-parallel execution model
  - Executes a native C/C++ function not compiled using the OpenCL compiler
  - This mode does not use a kernel object so arguments must be passed in
Command-Queues

- Command-queue execution
  - Execution model signals when commands are complete or data is ready
  - Command-queue could be explicitly flushed to the device
  - Command-queues execute in-order or out-of-order
    » In-order - commands complete in the order queued and correct memory is consistent
    » Out-of-order - no guarantee when commands are executed or memory is consistent without synchronization
Synchronization

- **Synchronization**
  - Signals when commands are completed to the host or other commands in queue
  - Blocking calls
    - Commands that do not return until complete
    - `clEnqueueReadBuffer()` can be called as blocking and will block until complete
  - **Event objects**
    - Tracks execution status of a command
    - Some commands can be blocked until event objects signal a completion of previous command
      - `clEnqueueNDRangeKernel()` can take an event object as an argument and wait until a previous command (e.g., `clEnqueueWriteBuffer`) is complete

- Queue barriers - queued commands that can block command execution
size_t global_work_size[1] = n; // set work-item dimensions
// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel,
   1,          // Work dimensions
   NULL,       // must be NULL (work offset)
   global_work_size,
   NULL,       // automatic local work size
   0,          // no events to wait on
   NULL,       // event list
   NULL);      // event for this kernel

// read output array
err = clEnqueueReadBuffer( context, memobjs[2],
   CL_TRUE,     // blocking
   0,           // offset
   n*sizeof(cl_float), // size
   dst,         // pointer
   0, NULL, NULL); // events
OpenCL C for Compute Kernels

• Derived from ISO C99
  - A few restrictions: recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported

• Built-in Data Types
  - Scalar and vector data types, Pointers
  - Data-type conversion functions: convert_type<_sat>_roundingmode>
  - Image types: image2d_t, image3d_t and sampler_t

• Built-in Functions — Required
  - work-item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions

• Built-in Functions — Optional
  - double precision, atomics to global and local memory
  - selection of rounding mode, writes to image3d_t surface
OpenCL C Language Highlights

• Function qualifiers
  - "__kernel" qualifier declares a function as a kernel
  - Kernels can call other kernel functions

• Address space qualifiers
  - __global, __local, __constant, __private
  - Pointer kernel arguments must be declared with an address space qualifier

• Work-item functions
  - Query work-item identifiers
    » get_work_dim(), get_global_id(), get_local_id(), get_group_id()

• Synchronization functions
  - Barriers - all work-items within a work-group must execute the barrier function before any work-item can continue
  - Memory fences - provides ordering between memory operations
OpenCL C Language Restrictions

- Pointers to functions are not allowed
- Pointers to pointers allowed within a kernel, but not as an argument
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported
- Writes to a pointer of types less than 32-bit are not supported
- Double types are not supported, but reserved
__kernel void vec_add (__global const float *a,
                 __global const float *b,
                 __global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
Legal Disclaimer

- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.

- Intel may make changes to specifications and product descriptions at any time, without notice.
- All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
- Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.
- Larrabee and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user.
- Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.
- Intel, Intel Inside and the Intel logo are trademarks of Intel Corporation in the United States and other countries.
- *Other names and brands may be claimed as the property of others.
- Copyright © 2009 Intel Corporation.
Risk Factors

This presentation contains forward-looking statements that involve a number of risks and uncertainties. These statements do not reflect the potential impact of any mergers, acquisitions, divestitures, investments or other similar transactions that may be completed in the future. The information presented is accurate only as of today’s date and will not be updated. In addition to any factors discussed in the presentation, the important factors that could cause actual results to differ materially include the following: Demand could be different from Intel’s expectations due to factors including changes in business and economic conditions, including conditions in the credit market that could affect consumer confidence; customer acceptance of Intel’s and competitors’ products; changes in customer order patterns, including order cancellations; and changes in the level of inventory at customers. Intel’s results could be affected by the timing of closing of acquisitions and divestitures. Intel operates in intensely competitive industries that are characterized by a high percentage of costs that are fixed or difficult to reduce in the short term and product demand that is highly variable and difficult to forecast. Revenue and the gross margin percentage are affected by the timing of new Intel product introductions and the demand for and market acceptance of Intel’s products; actions taken by Intel’s competitors, including product offerings and introductions, marketing programs and pricing pressures and Intel’s response to such actions; Intel’s ability to respond quickly to technological developments and to incorporate new features into its products; and the availability of sufficient supply of components from suppliers to meet demand. The gross margin percentage could vary significantly from expectations based on changes in revenue levels; product mix and pricing; capacity utilization; variations in inventory valuation, including variations related to the timing of qualifying products for sale; excess or obsolete inventory; manufacturing yields; changes in unit costs; impairments of long-lived assets, including manufacturing, assembly/test and intangible assets; and the timing and execution of the manufacturing ramp and associated costs, including start-up costs. Expenses, particularly certain marketing and compensation expenses, vary depending on the level of demand for Intel’s products, the level of revenue and profits, and impairments of long-lived assets. Intel is in the midst of a structure and efficiency program that is resulting in several actions that could have an impact on expected expense levels and gross margin. Intel’s results could be impacted by adverse economic, social, political and physical/infrastructure conditions in the countries in which Intel, its customers or its suppliers operate, including military conflict and other security risks, natural disasters, infrastructure disruptions, health concerns and fluctuations in currency exchange rates. Intel’s results could be affected by adverse effects associated with product defects and errata (deviations from published specifications), and by litigation or regulatory matters involving intellectual property, stockholder, consumer, antitrust and other issues, such as the litigation and regulatory matters described in Intel’s SEC reports. A detailed discussion of these and other factors that could affect Intel’s results is included in Intel’s SEC filings, including the report on Form 10-Q for the quarter ended June 28, 2008.