OpenCL and the quest for portable performance

OpenCL

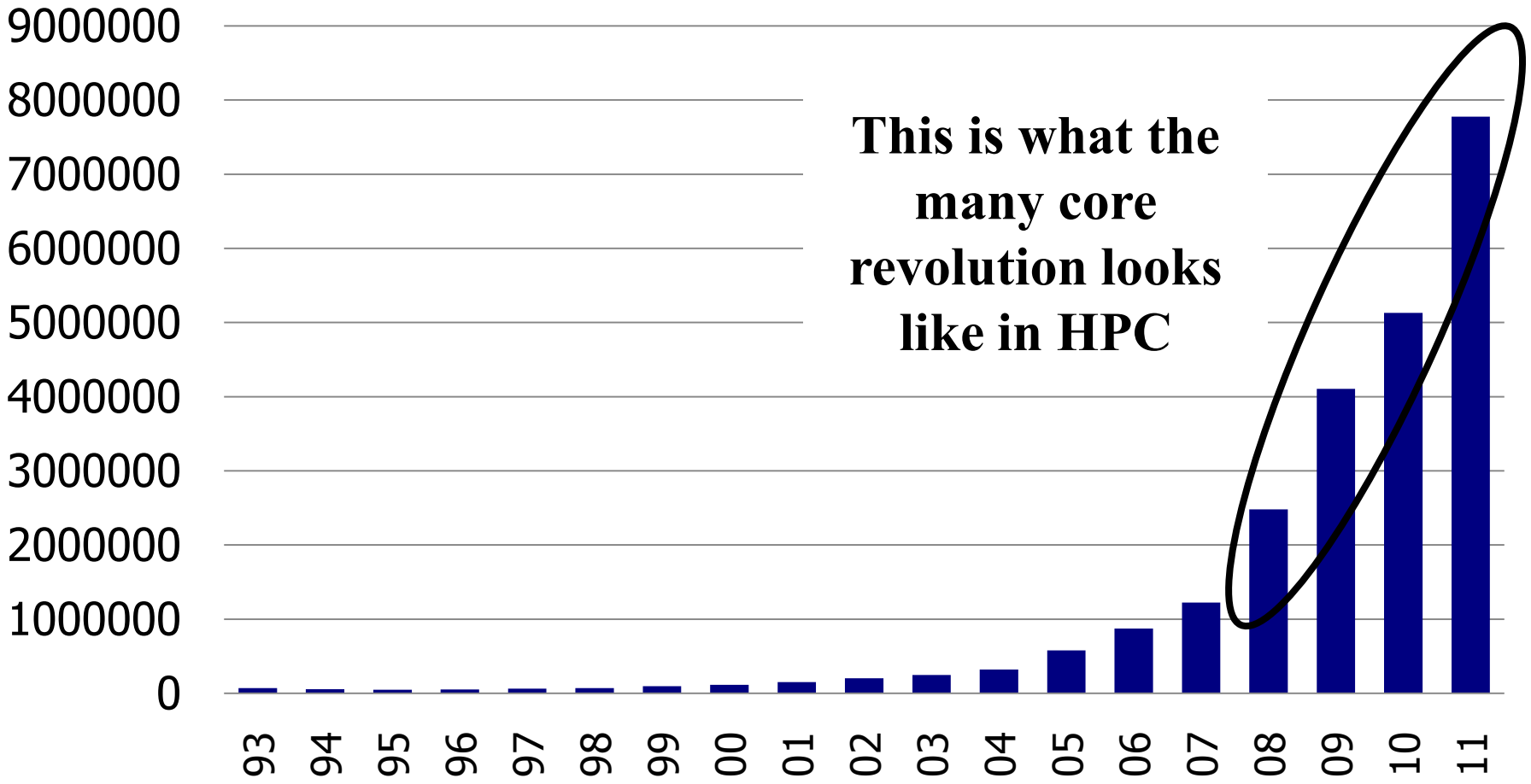
Tim Mattson
Intel Labs

Disclaimer

- The views expressed in this talk are those of the speaker and not his employer.
- I am in a research group and know very little about Intel products. So anything I say about them is not to be trusted.

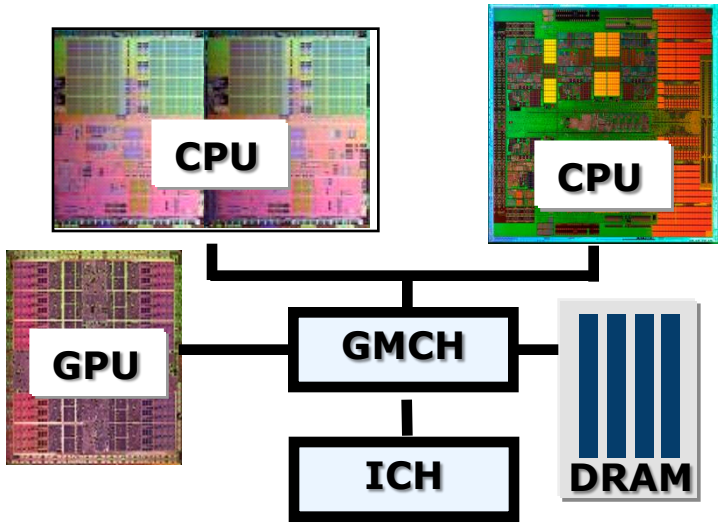
Parallel Hardware Trends – part 1

Top 500 supercomputers: total number of processors (1993-2011)



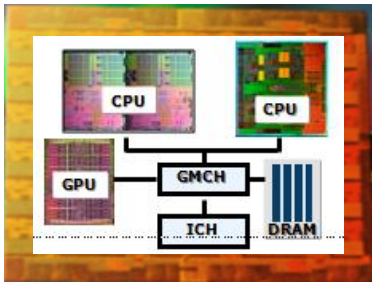
Source: the "June lists" from www.top500.org

Parallel Hardware Trends – part 2



- A modern platform has:
  - CPU(s)
  - GPU(s)
  - DSP processors
  - … other?

- And System on a Chip (SOC) trends are putting this all onto one chip



The future belongs to heterogeneous, many core SOC as the standard building block of computing

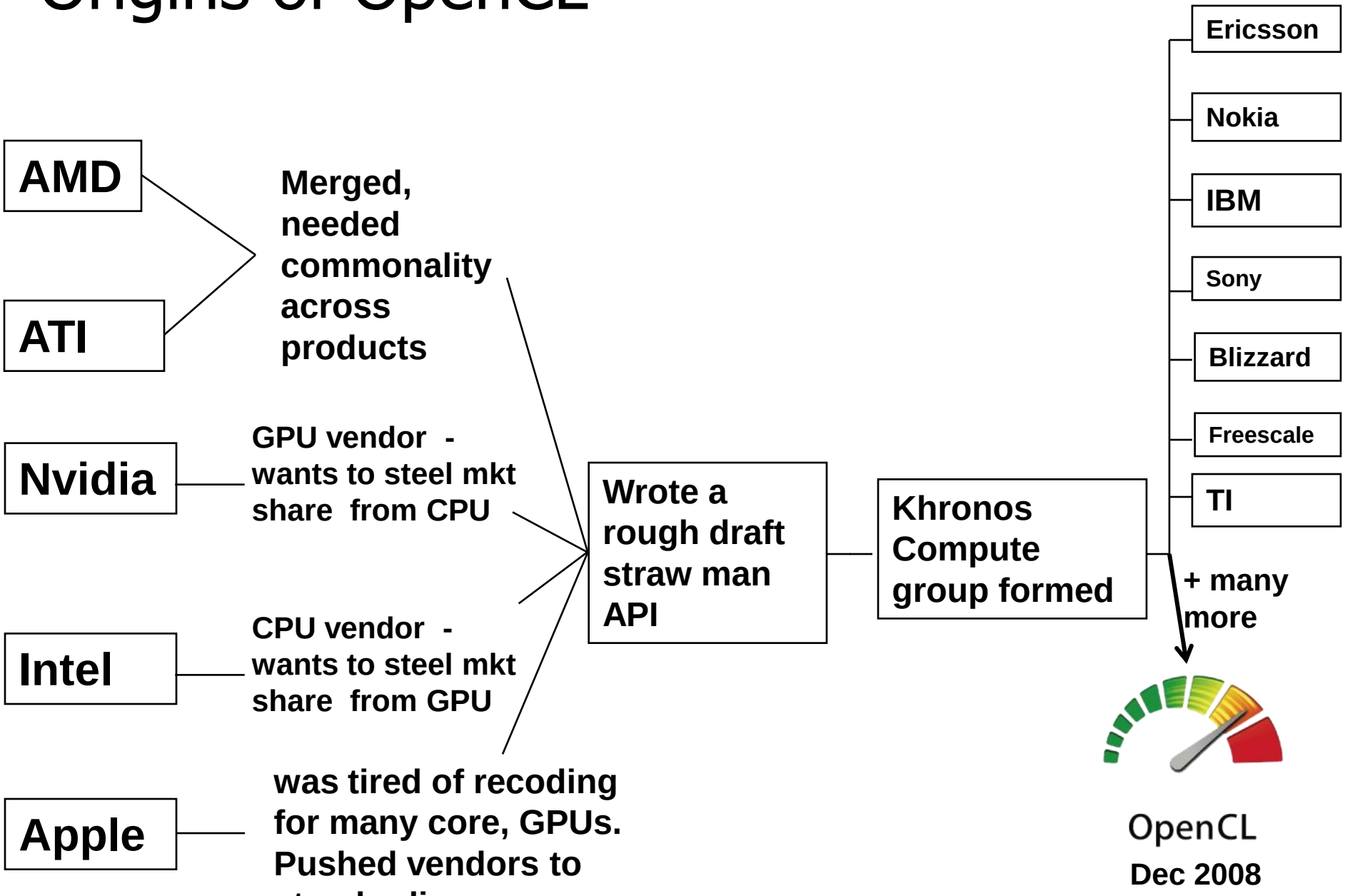
GMCH = graphics memory control hub, ICH = Input/output control hub SOC = system on a chip

The many core challenge

- A harsh assessment …
  - We have turned to multi-core chips not because of the success of our parallel software but because of our failure to continually increase CPU frequency.
- Result: a fundamental and dangerous mismatch
  - Parallel hardware is ubiquitous.
  - Parallel software is rare
- The Many Core challenge …
  - Parallel software must become as common as parallel hardware
  - Programmers need to make the best use of all the available resources from within a single program:

OpenCL is an industry standard attempt to address the many core challenge for heterogeneous systems

Origins of OpenCL



Agenda

- A brief overview of OpenCL
- OpenCL, the CPU and Performance portability
- The Future of OpenCL

OpenCL: a Standard for heterogeneous computing

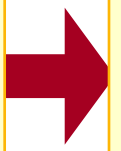
- Inspired by success with GPGPU programming, OpenCL is a standard that spans the full range of heterogeneous many core systems.
- OpenCL became an important standard “on release” by virtue of the market coverage of the companies behind it.

The BIG idea behind OpenCL

- OpenCL execution model … execute a kernel at each point in a problem domain.
  - E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

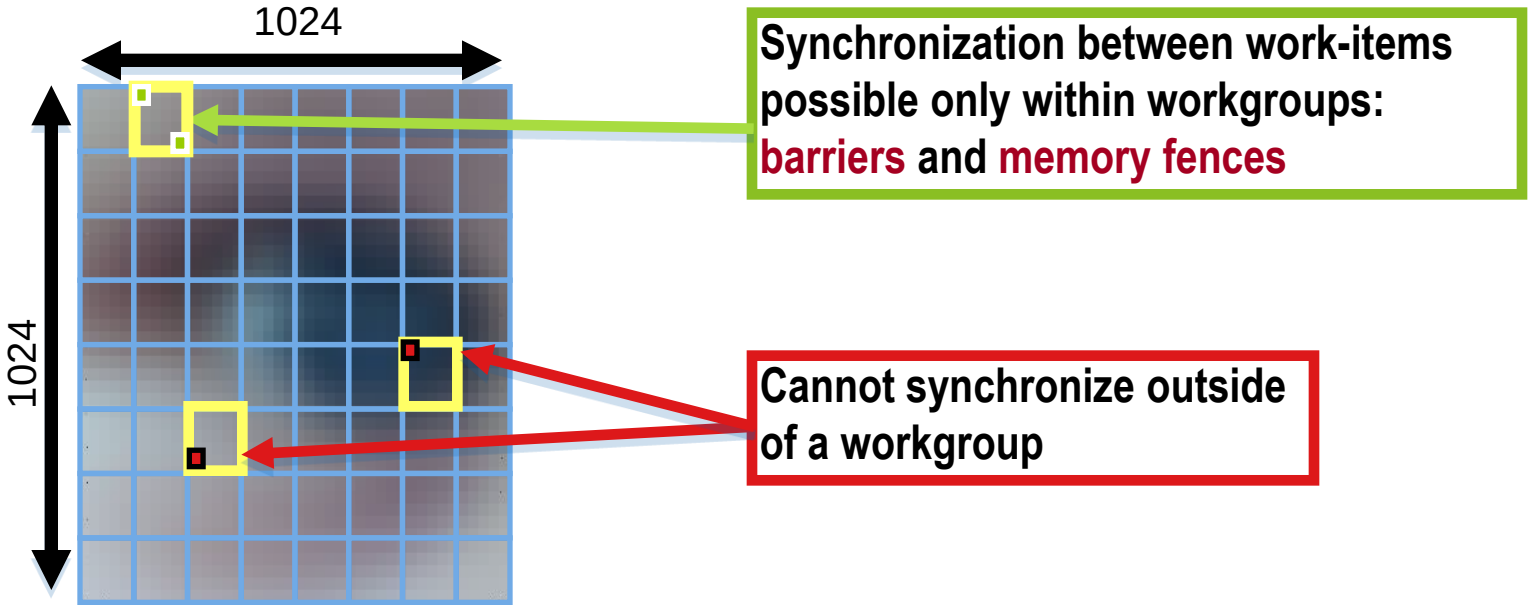


Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

An N-dimension domain of work-items

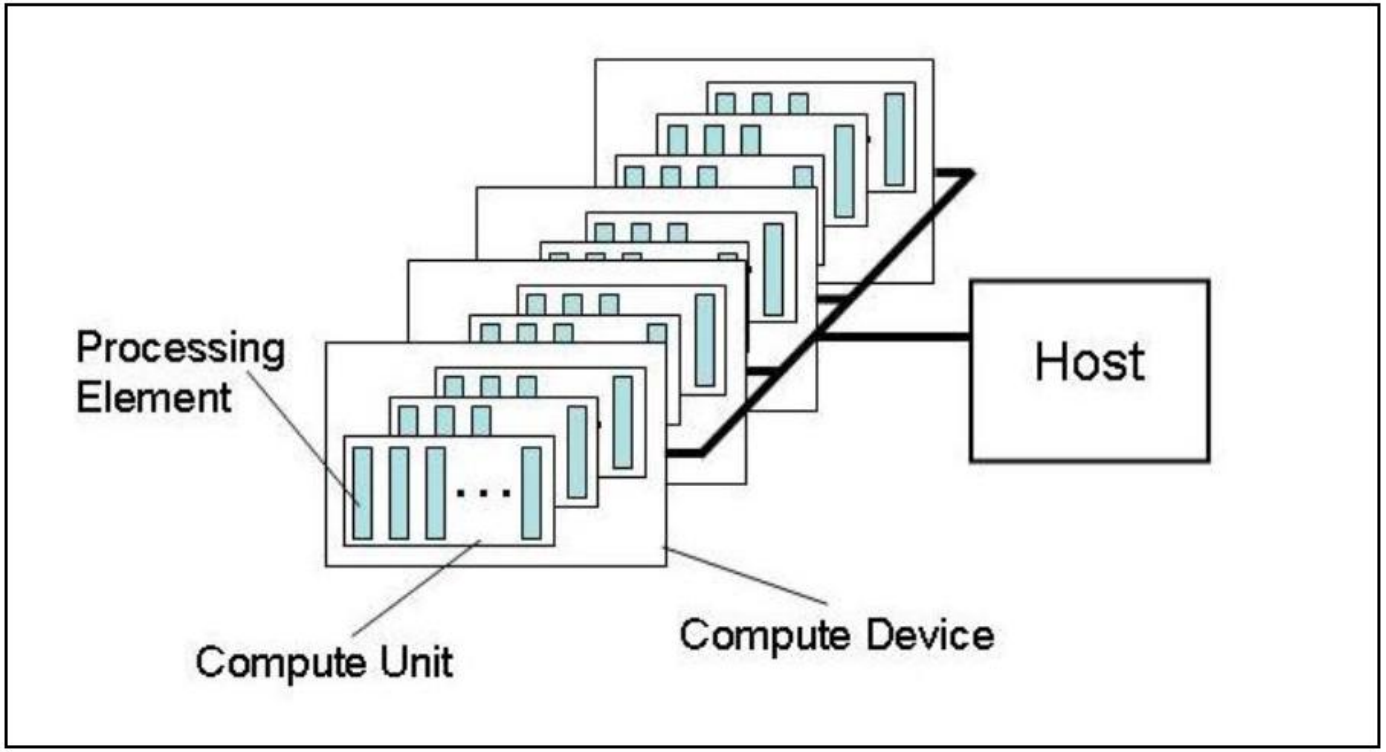
- Define an N-dimensioned index space that is "best" for your algorithm
  - Global Dimensions: 1024 x 1024 (whole problem space)
  - Local Dimensions: 128 x 128 (work group … executes together)



To use OpenCL, you must

- Define the platform
- Execute code on the platform
- Move data around in memory
- Write (and build) programs

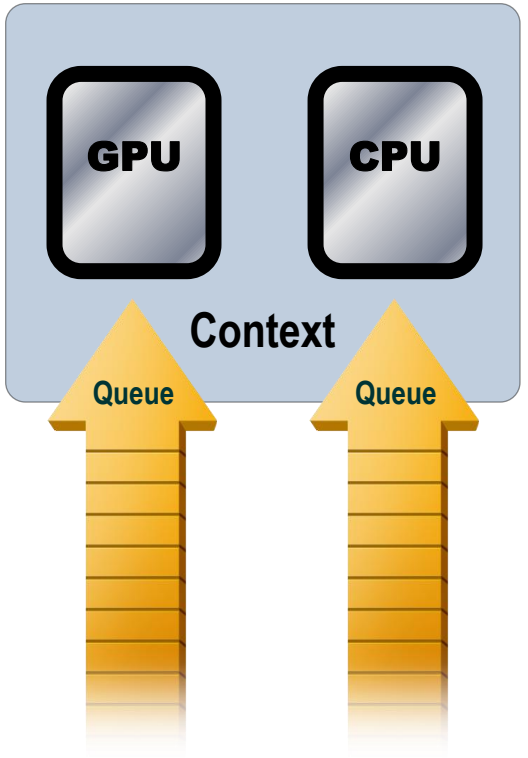
OpenCL Platform Model



- One Host + one or more Compute Devices
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements

OpenCL Execution Model

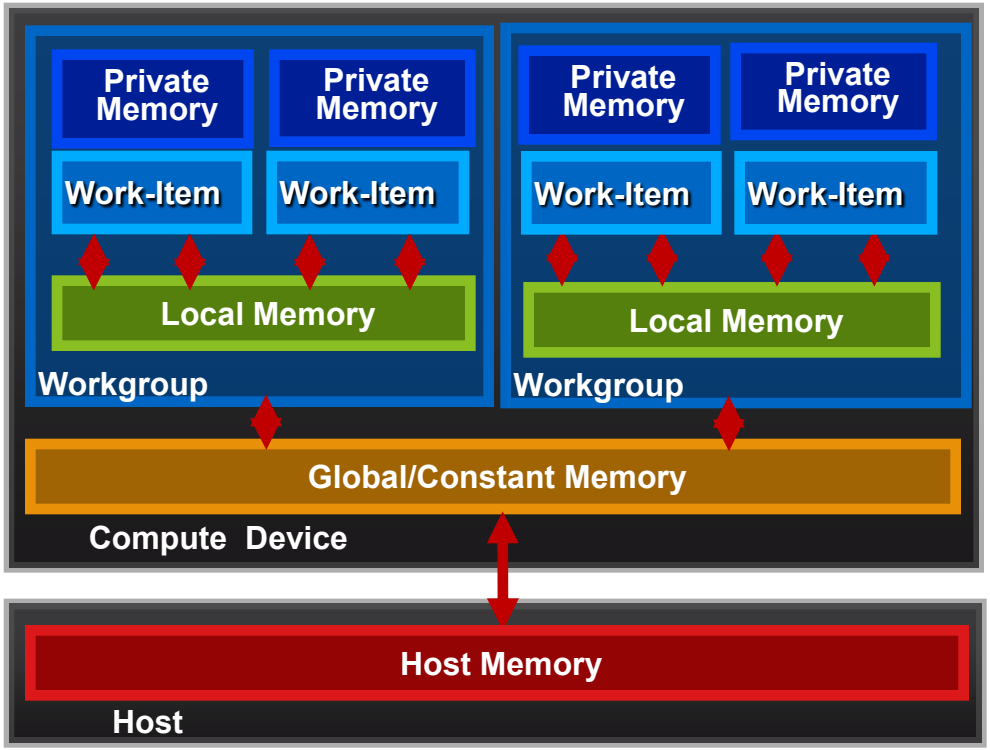
- An OpenCL application runs on a host which submits work to the compute devices.
  - **Work item**: the basic unit of work on an OpenCL device.
  - **Kernel**: the code for a work item. Basically a C function
  - **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
  - **Context**: The environment within which work-items executes … includes devices and their memories and command queues.



- Applications queue kernel execution instances
  - Queued in-order … one queue to a device
  - Executed **in-order** or **out-of-order** depending on queue-type

OpenCL Memory Model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU



Memory management is Explicit

You must move data from host -> global -> local … *and* back

Programming kernels:
the OpenCL C Language

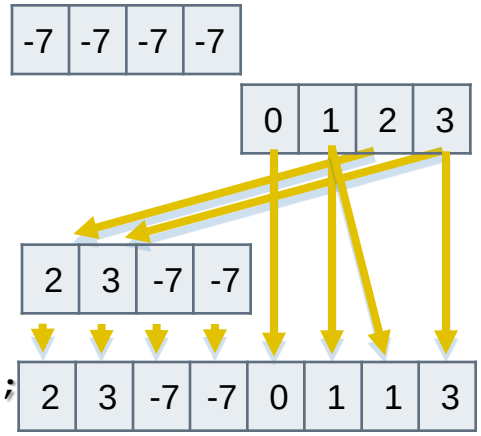
- A subset of ISO C99
  - But without some C99 features such as standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
- A superset of ISO C99 with additions for:
  - Work-items and workgroups
  - Vector types
  - Synchronization
  - Address space qualifiers
- Also includes a large set of built-in functions for image manipulation, work-item manipulation, specialized math routines, etc.

Programming Kernels: Data Types

- Scalar data types
  - char , uchar, short, ushort, int, uint, long, ulong, float
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)
- Image types
  - image2d_t, image3d_t, sampler_t
- Vector data types
  - Vector lengths 2, 4, 8, & 16 (char2, ushort4, int8, float16, double2, …)
  - Endian safe
  - Aligned at vector length
  - Vector operations and built-in functions

Double is an optional type in OpenCL 1.0

```
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);

vi0.lo = vi1.hi;

int8 v8 = (int8)(vi0, vi1.s01, vi1.odd);
```

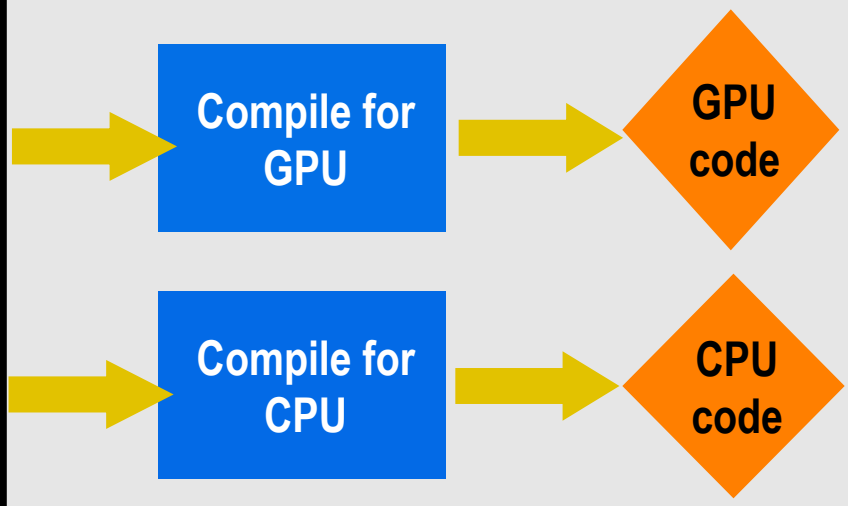


Building Program objects

- The program object encapsulates:
  - A context
  - The program source/binary
  - List of target devices and build options
- The Build process … to create a program object
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()

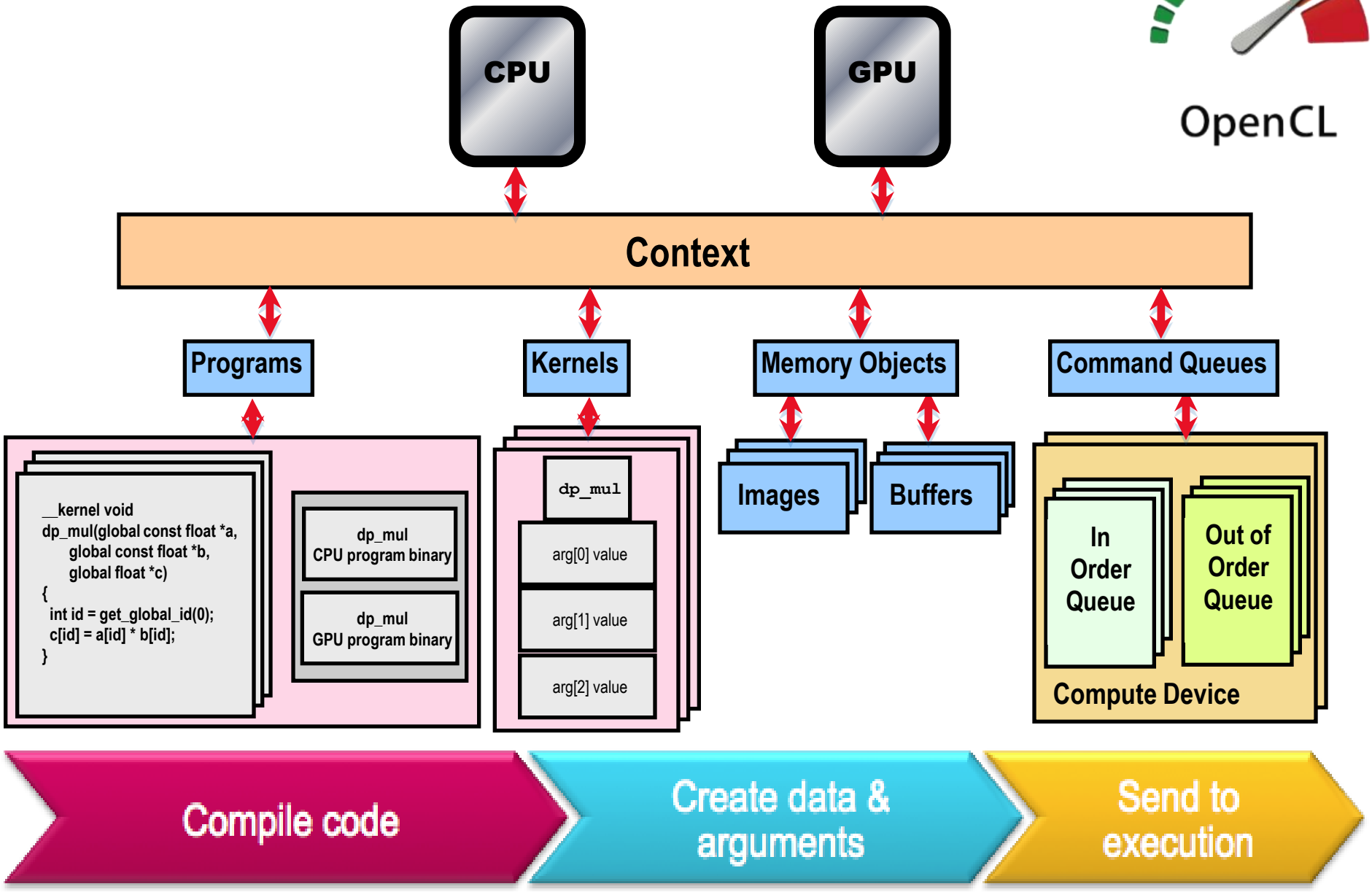
Kernel Code

```
kernel void
horizontal_reflect(
              read_only image2d_t src,
              write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src,
         sampler,(int2)(width-1-x, y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```



OpenCL and the Host Program

OpenCL



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
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);

// 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(cmd_queue,
   memobjs[2], CL_TRUE, 0, n*sizeof(cl_float),
   dst, 0, NULL, NULL);
```

Vector Addition - Host Program

Define platform and queues

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
// get the list of GPU devices
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);
```

Define Memory objects

```
// allocate the buffer memory objects
                                    ST_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

```
// crea
progra                                  ntext,
   1, &
```

Build the program

```
// Build the program
err =                                   , NULL, NULL,
   NULL, NULL);
```

Create and setup kernel

```
// create the kernel
k                                       add",

// 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));
```

Execute the kernel

```
// set work-item dimensions
global_

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

Read results on the host

```
// re
err =
   memobjs[2], CL_TRUE, 0, n*sizeof(cl_float),
   dst, 0, NULL, NULL);
```

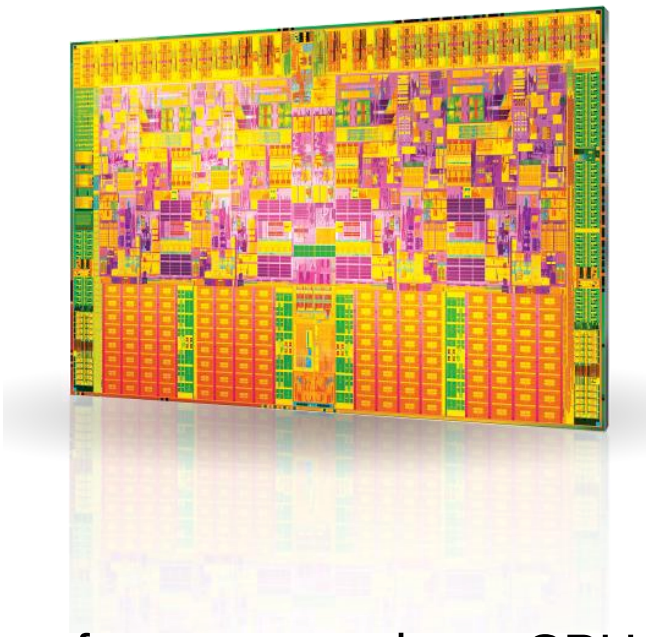
It's complicated, but most of this is "boilerplate" and not as bad as it looks.

Agenda

- A brief overview of OpenCL
- OpenCL, the CPU and Performance portability
- The Future of OpenCL

Heterogeneous computing and the CPU

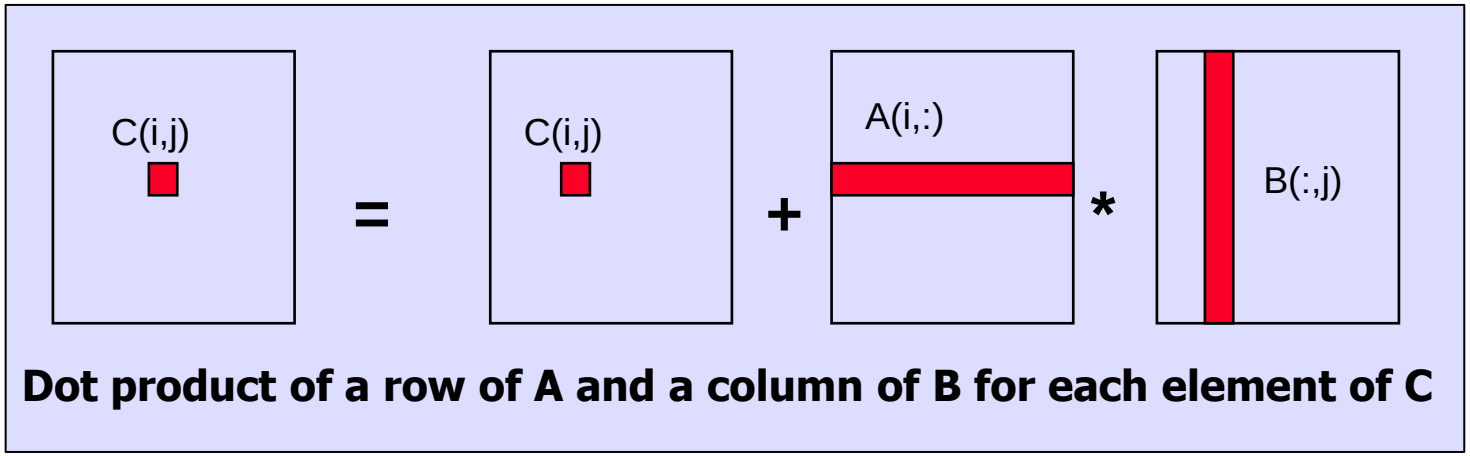
- Challenge … how do you exploit the performance of modern CPU's
  - Multi-Core
  - SMT
  - Vector Units



- Hypothesis: OpenCL is an effective platform for programming a CPU
  - OpenCL can handle multiple cores, multiple CPUs, and vector units.
  - Uses a single programming model which simplifies programming.
  - OpenCL provides a portable interface to vector instructions (SSE, AVX, etc).
  - The long term hope … Performance portable across CPU product lines and eventually between CPU and GPUs.

Matrix Multiplication: Sequential code

```
void mat_mul(int Mdim, int Ndim, int Pdim, float *A, float *B, float *C)
{
   int i, j, k;
   for (i=0; i<Ndim; i++){
      for (j=0; j<Mdim; j++){
         for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
            C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
         }
      }
   }
}
```



Matrix Multiplication: OpenCL kernel

```
__kernel mat_mul(
   const int Mdim, const int Ndim, const int Pdim,
 __global float *A, __global float *B, __global float *C)
{
   int i, j, k;
  i = get_global_id(0);
  j = get_global_id(1);
        for(k=0;k<Pdim;k++){      //C(i,j) = sum(over k) A(i,k) * B(k,j)
           C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
        }


}
```

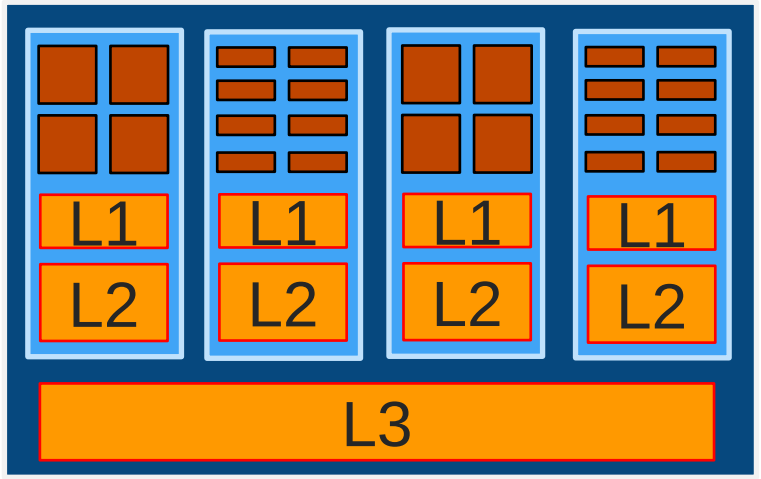
Matrix Multiplications Performance

- Results on an Apple laptop with an NVIDIA GPU and an Intel CPU. Matrices are stored in global memory.
- No effort AT ALL was done to optimize. This is naïve "out of the box" performance.

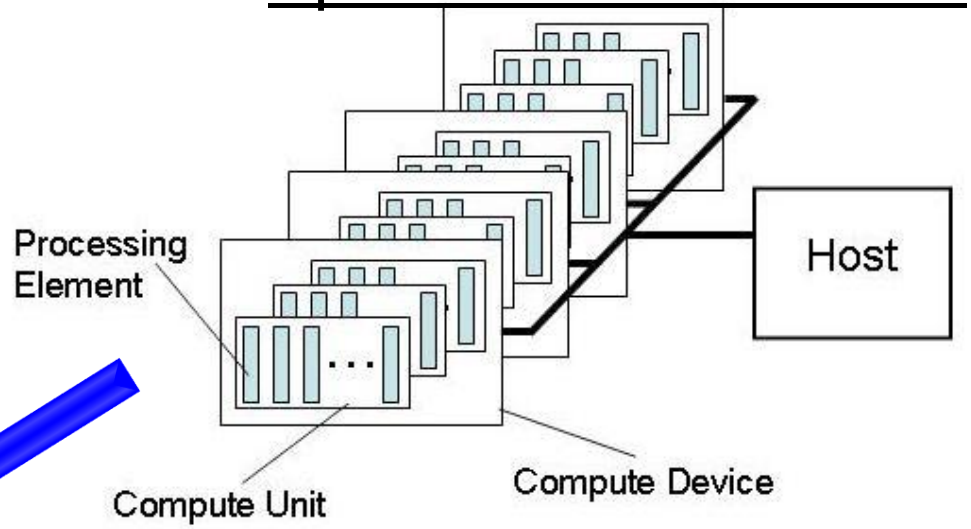
Case	MFLOPS
CPU: Sequential C (not OpenCL)	167
GPU: C(i,j) per work item, all global	511
CPU: C(i,j) per work item, all global	744

Device is GeForce® 8600M GT GPU from NVIDIA with a max of 4 compute units
Device is Intel® Core™2 Duo CPU T8300 @ 2.40GHz

OpenCL view of Core™ i7



OpenCL Platform Model*



Core™ i7 975

- **8 Compute Units (CU)**
  - **Quad Core + SMT**
- **4/8/16 Processing Elements (PE) per CU**
  - **128bit XMM registers**
  - **Data type determines # of elements…**
- **(32K L1 + 256K L2) Per PE, 8M L3 Local mem.**

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

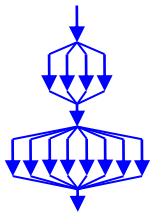
Explicit Data Parallelism on the CPU

- Workitem is executed solely on a single compute unit (HW Thread)
- OpenCL Vector operations are mapped to SSE instructions

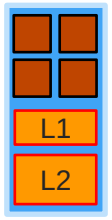
- Workgroup is executed on a single compute unit (HW Thread)

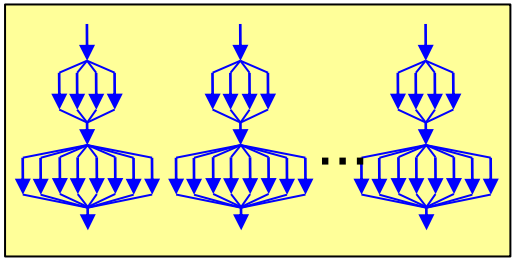
- Kernel is executed over an N-D Range, which is divided into workgroups
- Several Workgroups run concurrently on all compute unit (HW threads)

Kernel Execution **OCL Device**

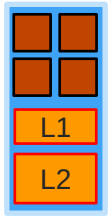


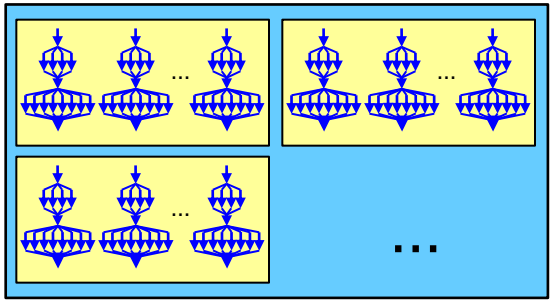
Workitem



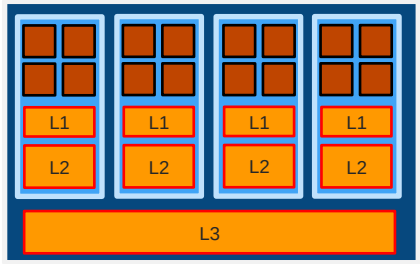


Workgroup





N-D Range



Explicit SIMD data parallelism

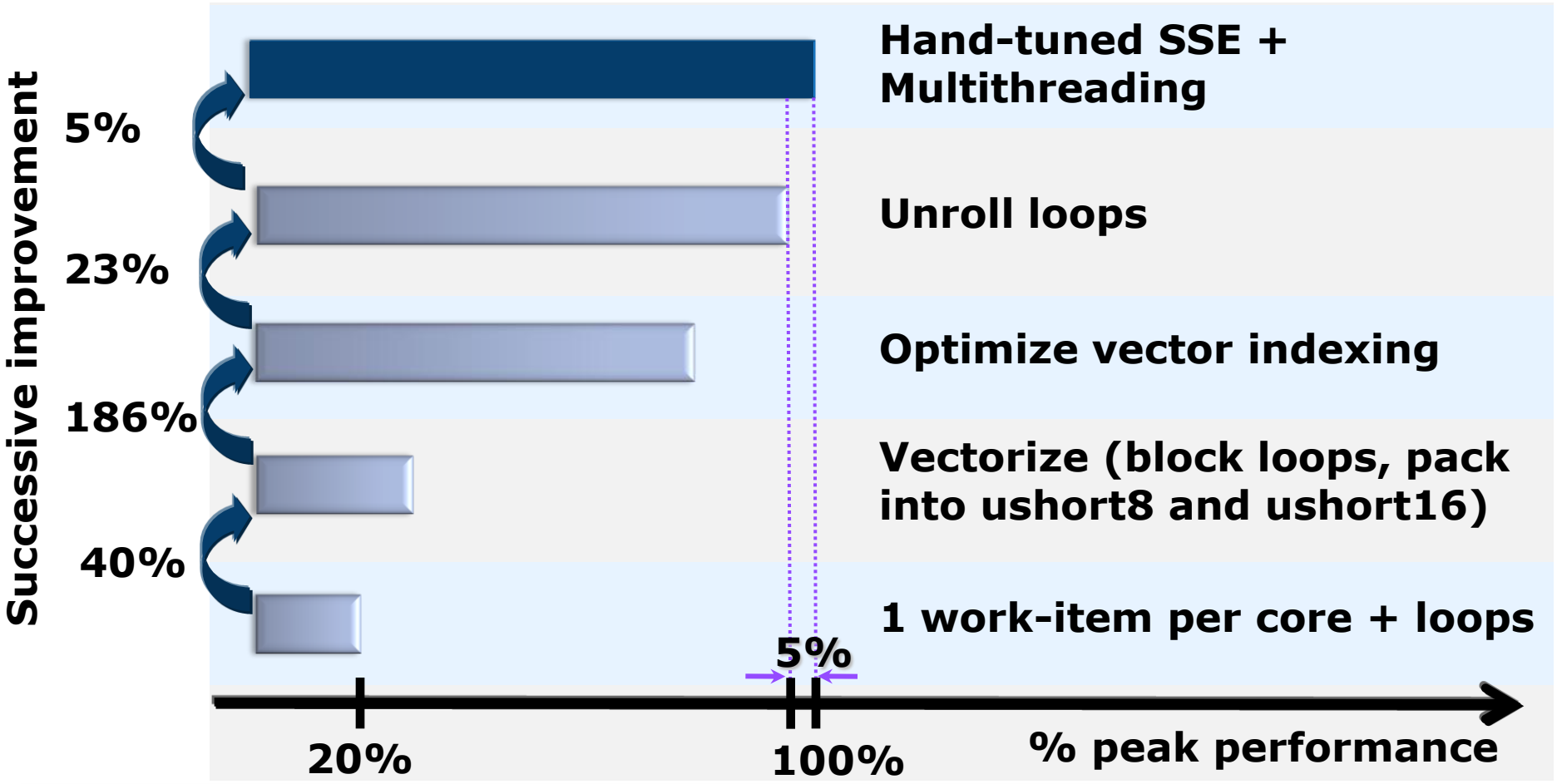
- OpenCL as a portable interface to vector instruction sets.
  - Block loops and pack data into vector types (float4, ushort16, etc).
  - Replace scalar ops in loops with blocked loops and vector ops.
  - Unroll loops, optimize indexing to match machine vector width

```
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

Explicit SIMD data parallelism: Case Study

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

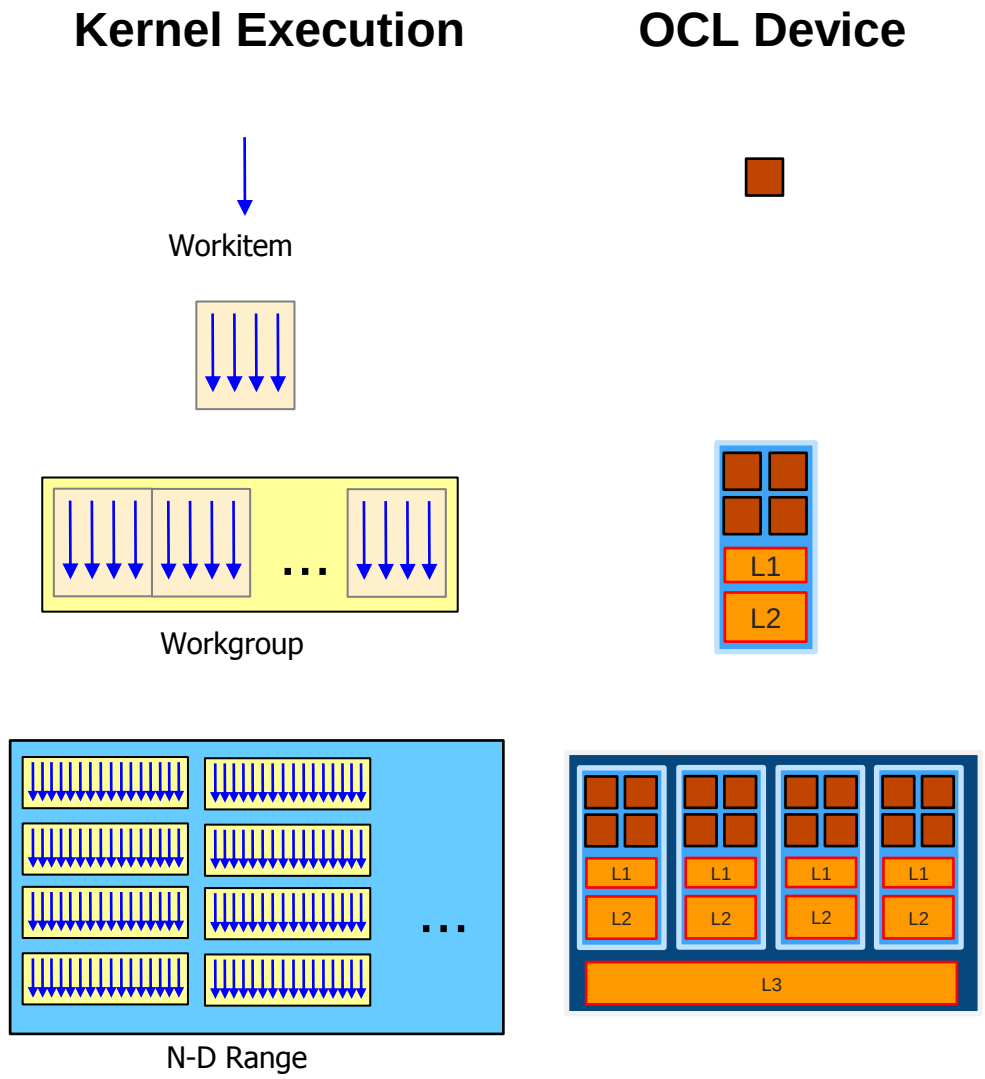


Good news: OpenCL code 95% of hand-tuned SSE/MT perf.

Bad news: New platform, redo all those optimizations.

Implicit Data Parallelism on the CPU

- One workitem runs on a single SSE lane
- Workitems are packed to SSE registers as part of the OpenCL Compilation process
- Workgroup is executed on a compute unit (HW Thread)
- Kernel is executed over an N-D Range, which is divided to workgroups
- Several Workgroups run concurrently on all compute unit (HW threads)



Implicit vs. explicit SIMD: N-body Simulation

Given *N* bodies with an initial position x_i and velocity v_i for, the force f_{ij} on body *i* caused by body *j* is given by following (G is gravity):

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{\left\|\mathbf{r}_{ij}\right\|^2}\cdot\frac{\mathbf{r}_{ij}}{\left\|\mathbf{r}_{ij}\right\|}, \qquad \mathbf{F}_i = \sum_{\substack{1\le j\le N \\ j\neq i}} \mathbf{f}_{ij}$$

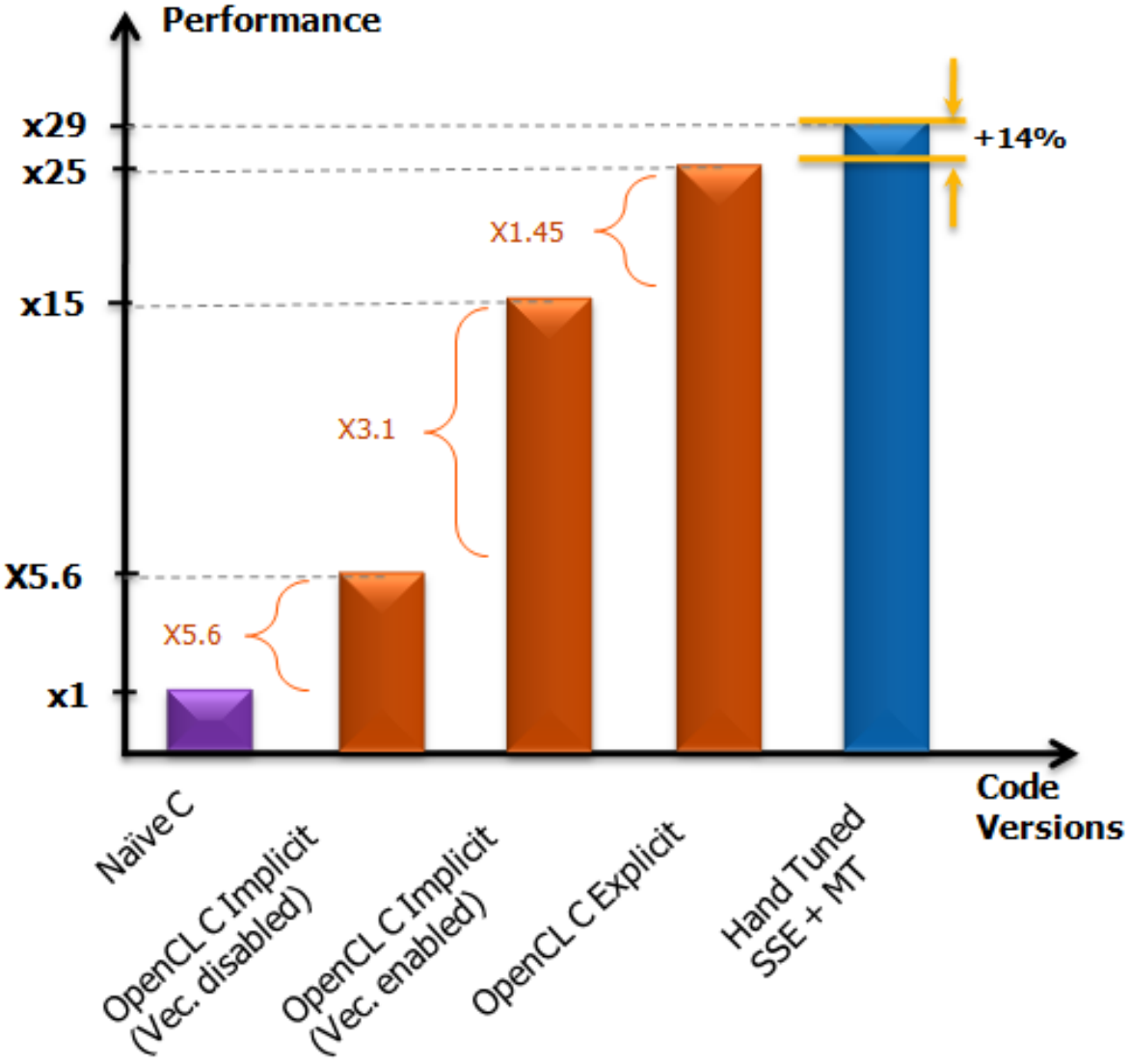
where m_i and m_j are the masses of bodies *i* and *j*, respectively; $r_{ij} = x_j - x_i$

The acceleration is computed as $a_i = F_i/m_i$

NBody Performance

Results from Intel's internal OpenCL implementation:*

- Implicit Data Parallelism
  - "shader-style" code
  - Benefit from multi-core/SMT
- Explicit Data-Parallelism
  - Hand tuned OpenCL C code
  - OpenCL Explicit version is x25 faster than Naïve C *
  - Explicit version is only 14% slower than highly optimized code *



* Results measured on Core™ i7 975, 3.3 GHz, 6GB DDR3
Results depends on the algorithm/code running

Agenda

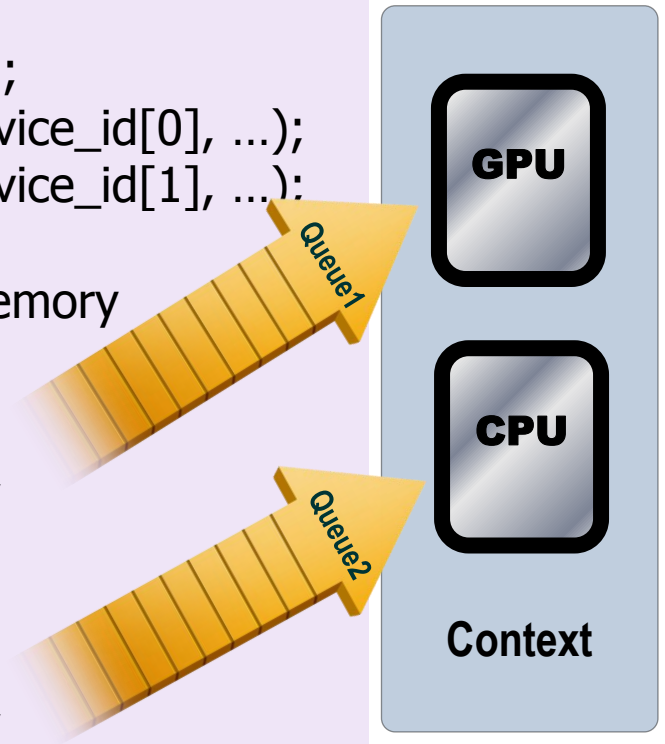
- A brief overview of OpenCL
- OpenCL, the CPU and Performance portability
- The Future of OpenCL

Acknowledgements/disclaimer: These slides are based on conversations with the OpenCL team at Intel. They give you insights into our thinking about what needs to happen to OpenCL. But we may never propose these constructs to Khronos or if we do, the actual proposals may look totally different.

Balancing work between multiple devices

- Currently, the programmer must split the work explicitly between devices … managing work distribution, memory objects, and other details “by hand”.

```
// The Host program
device[0] = <<< A GPU >>>
device[1] = <<< A CPU >>>
context = clCreateContext(0, 2, &device_id, … );
Queue1 = clCreateCommandQueue(context, device_id[0], …);
Queue2 = clCreateCommandQueue(context, device_id[1], …);

////  build programs, set up kernels, manage memory

for(i=0;i=NwrkGPU;i++){
    err = clEnqueueNDRangeKernel(Queue1, …);
}

for(i=0;i=NwrkCPU;i++){
    err = clEnqueueNDRangeKernel(Queue2, …);
}
```

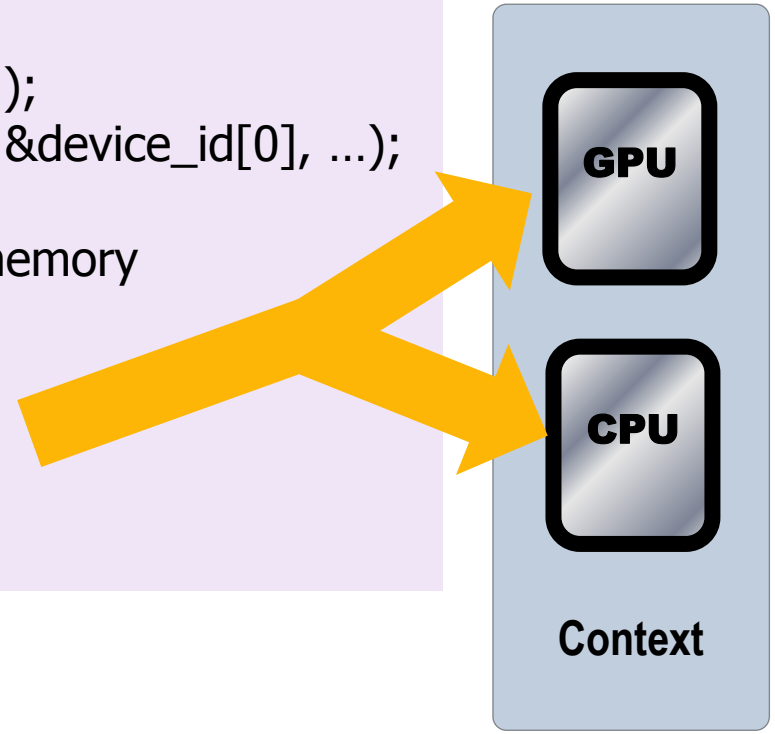


- Problem: you generally don’t know the runtime-per-kernel so any “hand decomposition” is likely to be poor.

Balancing work between multiple devices

- Solution : One queue that serves multiple devices

```
// The Host program
device[0] = <<< A GPU >>>
device[1] = <<< A CPU >>>
context = clCreateContext(0, 2, &device_id, … );
Queue  = clCreateCommandQueue(context, 2, &device_id[0], …);

////  build programs, set up kernels, manage memory

for(i=0;i=NwrkGPU+NwrkCPU;i++){
    err = clEnqueueNDRangeKernel(Queue, …);
}
```



- Managing the memory objects can be difficult. It would be nice to have a shared address space between the devices and the host

SVM: Shared Virtual Memory

- Shared Virtual Memory (SVM)
  - An address space shared between the host and one or more devices so they can all work on same objects at one time and through pointers.
  - Use OpenCL's current "object based" release consistency.
- Restricted to race free programs:
  - Read-Read conflicts OK
  - Read-Write conflicts OK with careful synchronization
  - Write-write conflicts not allowed
- Changes to OpenCL
  - Add a new data access qualifier __shared
  - For example, kernel code might look like:

```
__kernel foo (__shared const float *list) {
    __shared float *ptr;
    //  do a bunch of stuff with ptr
}
```

SVM + multi-device queues would dramatically improve the ability to write performance portable programs in OpenCL.

… but OpenCL is still “broken”

- The tasking model in OpenCL is weak.
- Here is what we need:
  - Tasks as the foundation of OpenCL
    - A task is a unit of work plus a data environment
    - We need a symmetric model:
      - Kernels applied across an NDRange are just a mechanism to create a static set of tasks.
  - Tasks can dynamically enqueue tasks … so tasks can create new tasks.

```
//  Enqueue to the same queue used by the parent task
event_t ev1 = enqueue(NULL)
              foreach(int i=0;i<512;i++) bar(i, buffer_B);

// Enqueue to a queue passed as a kernel arg to the parent task
event_t  ev2= enqueue(after:ev1; on:otherDevCmdQ)
              foreach(int i=0;i<512;i++) bar2(i, buffer_B);
```

OpenCL code needs to be easier to write

- Define a high level interface to OpenCL to make it easier to work with:
  - A source to source preprocessor that takes simple high level code and outputs correct OpenCL.
  - Define kernels with a keyword modifying a regular function … avoid the whole “load a kernel as string” step.
  - Hide the complexity of the Host platform and runtime APIs.
  - Do everything as a source to source translator so integration with full OpenCL is well defined.

Goal: Make the most common cases easy

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
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);

// 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(cmd_queue,
   memobjs[2], CL_TRUE, 0, n*sizeof(cl_float),
   dst, 0, NULL, NULL);
```

Vector Addition - Host Program

Define platform and queues

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
// get the list of GPU devices
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);
```

Define Memory objects

```
// allocate the buffer memory objects
   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

```
// cre
progra
   1, &
```

Build the program

```
//
er
   NULL, NULL);
```

Create and setup kernel

```
// create the kernel
k
// 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));
```

Execute the kernel

```
// set work-item dimensions
global_
// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue,
   kernel, 1, NULL, global_work_size, NULL, 0,
   NULL, NULL);
```

Read results on the host

```
// re
err =
   memobjs[2], CL_TRUE, 0, n*sizeof(cl_float),
   dst, 0, NULL, NULL);
```

It's complicated, but most of this is "boilerplate" and not as bad as it looks.

What might the code look like?

```
__kernel void sum(__global int* in_bufA, __global int* in_bufB,
                                         __global int* out_res)
{   int tid = get_global_id(0);
    out_result[tid] = in_bufA[tid] + in_bufB[tid];
}
int main(int argc, char* argv[])
{
  oclInit();
  cl_mem bufA=clCreateBuffer(GPU,CL_MEM_READ_WRITE,512,NULL,NULL);
  cl_mem bufB=clCreateBuffer(GPU,CL_MEM_READ_WRITE,512,NULL,NULL);
  cl_mem res =clCreateBuffer(GPU,CL_MEM_READ_WRITE,512,NULL,NULL);

  AddDataToBuffer (bufA, 512);
  AddDataToBuffer (bufB, 512);

  cl_event sumFinished = oclDispatch [512] <GPU> sum (myFooVec,
                                         bufA, bufB, result);
  PrintBuffer(result);

  oclRelease();
}
```

Summary: Enhancements to OpenCL

- Multi-device queues for dynamic load balancing
- Shared virtual memory:
  - To make sharing work across devices easier
  - To support kernels operating on pointer based data structures
- Fix the tasking model to generalize OpenCL to a wider range of algorithms
- Create a higher level “single source” interface to OpenCL that automates the verbose stuff and makes OpenCL easier to write.

Acknowledgements/disclaimer: These slides are based on conversations with the OpenCL team at Intel. They give you insights into our thinking about what needs to happen to OpenCL. But we may never propose these constructs to Khronos or if we do, the actual proposals may look totally different.

Conclusion

- OpenCL is an effective platform for programming a CPU.
- Implicit SIMD code suggests a route to portably performant code
- Our OpenCL technology needs more work to achieve this full potential of implicit SIMD, but note ... we know this approach works since its based on the way shader compilers work today.

I’d rather be surfing