OpenCL

Dr. David Brayford, LRZ, brayford@lrz.de

PRACE PATC: Intel MIC & GPU Programming Workshop

Open Computing Language

Open, royalty-free standard C-language extension

For cross-platform, parallel programming of modern processors

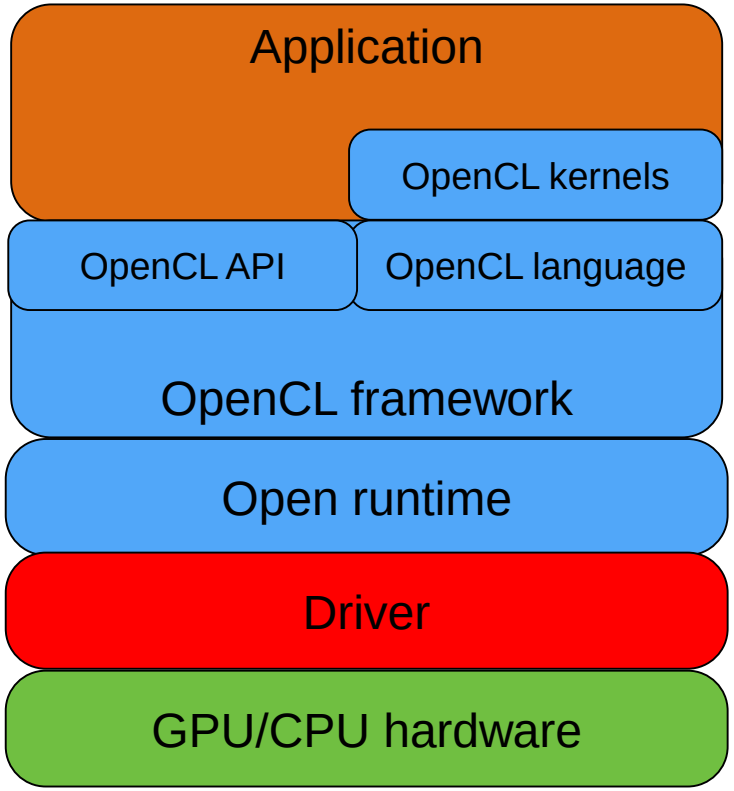
Initially proposed by Apple, and approved by Intel, Nvidia, AMD ...etc.

Specification by Khronos group (same group as OpenGL standard)

It intends to unify the access to heterogeneous hardware accelerators

- CPUs (Intel i7, AMD,…)
- GPUs (Nvidia GTX c & Tesla, AMD/ATI 58xx, …)

OpenCL architecture



OpenCL Basics

OpenCL uses runtime compilation, because it is not always possible to know details of the device that the kernel will execute upon.

Platforms: GPU, CPU, MIC ….

Kernel Code

Vectorized algorithm.

Host Code

Setup the environment for the OpenCL program

Create and manage kernels.

Context and Command-Queues

Context:

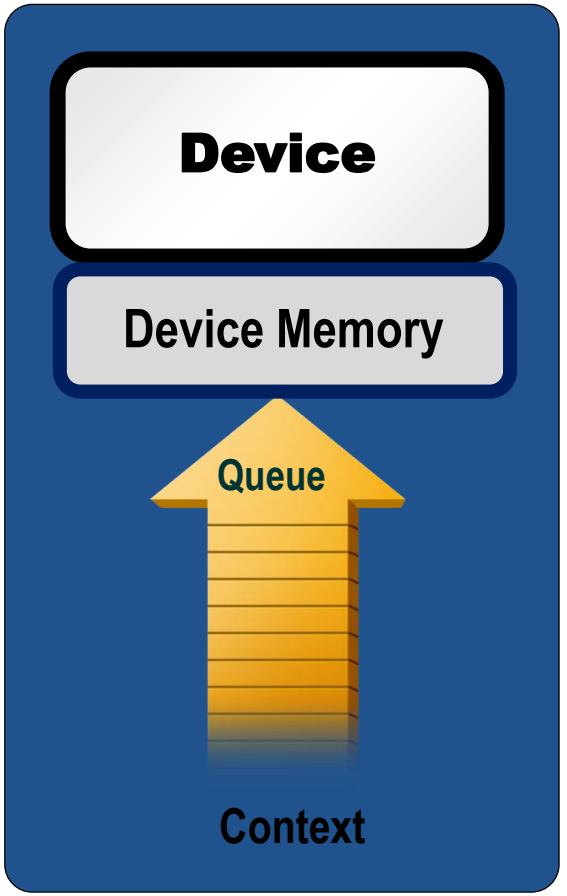
The environment within which kernels execute and in which synchronization and memory management is defined.

The **context** includes:

One or more devices
Device memory
One or more command-queues

All **commands** for a device (kernel execution, synchronization, and memory operations) are submitted through a **command-queue**.

Each **command-queue** points to a single device within a context.



Exercise 1:

Query the devices

OpenCL Host Program

The host program … the code that runs on the host:

Setup the environment for the OpenCL program
Create and manage kernels

Five simple steps in a basic host program

Define the platform … platform = devices + context + queues.

Create and Build the program (dynamic library for kernels).

Setup memory objects.

Define kernel (attach arguments to kernel function).

Submit commands … transfer memory objects and execute kernels.

Define the platform

Grab the first available Platform:

```
err = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
```

Use the first CPU device the platform provides:

```
err = clGetDeviceIDs(firstPlatformId, CL_DEVICE_TYPE_CPU, 1,
                                                &device_id, NULL);
```

Create a simple context with a single device:

```
context = clCreateContext(firstPlatformId, 1, &device_id, NULL,
                                                NULL, &err);
```

Create a simple command queue to feed our compute device:

```
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

Create the Program

Define source code for the kernel-program as a string literal (test programs) or read from a file (real applications).

the program object:

```
program = clCreateProgramWithSource(context, 1,
        (const char **) & KernelSource, NULL, &err);
```

Compile the program to create a "dynamic library" from which specific kernels can be pulled:

```
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

Fetch and print error messages: if (err != CL_SUCCESS) :

```
size_t len;          char buffer[2048];
clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
                                    sizeof(buffer), buffer, &len);
printf("%s\n", buffer);
```

Setup Memory Objects

For two vector addition, 3 memory objects … one for each input vector (A and B) and one for the output vector (C).

Create input vectors and assign values on the host:

```
float       h_a[LENGTH], h_b[LENGTH], h_c [LENGTH];
for(i = 0; i < count; i++) {
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
}
```

Define OpenCL memory objects:

```
d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count, NULL, NULL);
d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count, NULL, NULL);
d_c  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY, sizeof(float) * count, NULL, NULL);
```

Creating Buffers

Buffers are declared as type:

```
cl_mem a;
```

Define arrays in host memory to hold values you wish to put-in or get-from the buffer:

```
float a_p[LENGTH], c_p[LENGTH];
```

Create the buffer (a), assign sizeof(float)*count bytes from "a_p" to the buffer, and copy it into device memory:

```
cl_mem a = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(float) * count, a_p, NULL);
```

Other common memory flags include:

```
CL_MEM_WRITE_ONLY, CL_MEM_READ_WRITE
```

Submit a command to copy a buffer into host memory at "c_p":

```
clEnqueueReadBuffer(queue, cp_out, CL_TRUE, 0, sizeof(float) * count,
                                                cp_res, NULL, NULL, NULL);
```

Exercise 2:

Simple OpenCL in C

Define the Kernel

Create kernel object from the kernel function “vadd”:

```
kernel = clCreateKernel(program, "vadd", &err);
```

Attach arguments to the kernel function “vadd” to memory objects:

```
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &count);
```

Submit

Write Buffers from host into global memory (as non-blocking operations)

```
err = clEnqueueWriteBuffer( commands, d_a, CL_FALSE, 0,
                            sizeof(float) * count, h_a, 0, NULL, NULL );
err = clEnqueueWriteBuffer( commands, d_b, CL_FALSE, 0,
                            sizeof(float) * count, h_b, 0, NULL, NULL );
```

Enqueue the kernel for execution (note: in-order queue so this is OK)

```
err = clEnqueueNDRangeKernel( commands, kernel, 1, NULL,
                                        &global, &local, 0, NULL, NULL );
```

Read back the result (as a blocking operation). Use the fact that we have an in-order queue which assures the previous commands are done before the read begins.

```
err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0,
                           sizeof(float) * count, h_c, 0, NULL, NULL );
```

OpenCL 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 1.1 Language Restrictions

Pointers to functions are ***not*** allowed.

Pointers to pointers allowed within a kernel, but not as an argument. OpenCL 2.0 supports shared virtual pointers.

Bit-fields are not supported, not portable, ordering of bits in memory is hardware dependent.

Variable length arrays and structures are not supported.

Recursion is not supported.

Exercise 3:

OpenCL C++ Host and Separate Kernel File.

Matrix Multiplication C

We calculate C = AB, dimA=(N x P), dimB=(P x M)

```
void matrix_multiply(int Mdim, int Ndim, int Pdim, float *A, float *B, float *C)
{
  for (int I = 0; I < Ndim; i++)
  {
    for (int j = 0; j < Mdim; j++)
    {
      for (int 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];
      }
    }
  }
}
```

Dot product of a row of A and a column of B for each element of C

CPU : ~887 MFLOPS

Matrix Multiplication OpenCL

```
__kernel void mat_mul(const int Mdim, const int Ndim, const int Pdim,
                      __global float *A, __global float *B, __global float *C)
{
  // remove outer loops and set work items
  int i = get_global_id(0);
  int j = get_global_id(1);
  int k = 0;
  float tmp = 0.0f;
  for (k = 0; k < Pdim; k++)
  {
     // C[i * Ndim + j] += A[i * Ndim + k] * B[k * Pdim + j];
     // common optimization for matrix multiplication
     tmp += A[i*Ndim+k] * B[k*Pdim+j];
     C[i*Ndim+j] += tmp;
  }
}
```

CPU : ~3926.1 MFLOPS
GPU : ~3720.9 MFLOPS

Exercise 4:

Write OpenCL Kernel that element wise adds 3 vectors

Result = A + B + C

OpenCL memory model

Managing the memory hierarchy is one of the most important things to

Private Memory

Per work-item

Local Memory

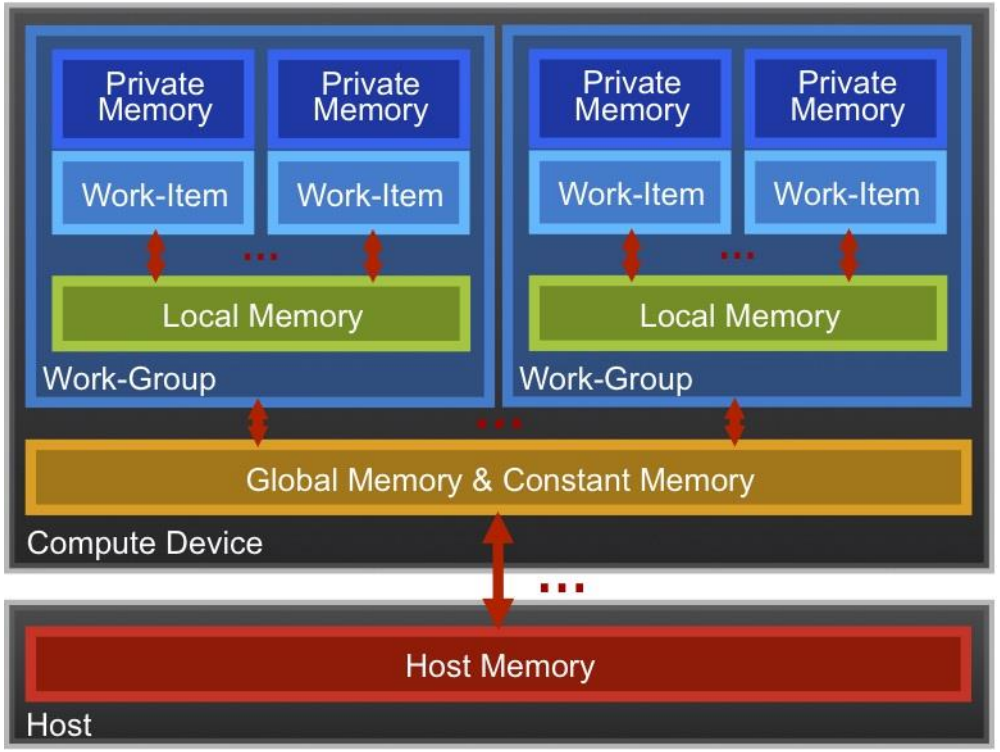
Shared within a work-group

Global/Constant Memory

Visible to all work-groups

Host Memory

On the CPU



Memory management is explicit:
You must move data from host -> global -> local *and* back

Private Memory

Private Memory:

A very scarce resource, only a few tens of 32-bit words per Work-Item at most.

If you use too much it spills to global memory or reduces the number of Work-Items that can be run at the same time, potentially harming occupancy.

Think of these like registers on the CPU.

Local Memory

Tens of KBytes per Compute Unit

As multiple Work-Groups will be running on each compute unit, this means only a fraction of the total Local Memory size is available to each Work-Group

Assume O(~1-10) KBytes of Local Memory per Work-Group

Your kernels are responsible for transferring data between Local and Global/Constant memories … there are optimized library functions to help E.g. async_work_group_copy(), async_workgroup_strided_copy(), …

Use Local Memory to hold data that can be used by all the Work-Items in a Work-Group

Access patterns to Local Memory affect performance in a similar way to accessing Global Memory

Have to think about things like coalescence, bank conflicts etc.

Memory Consistency

OpenCL uses a relaxed consistency memory model; i.e.

The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.

Within a work-item:

Memory has load/store consistency to its private view of memory.

Within a work-group:

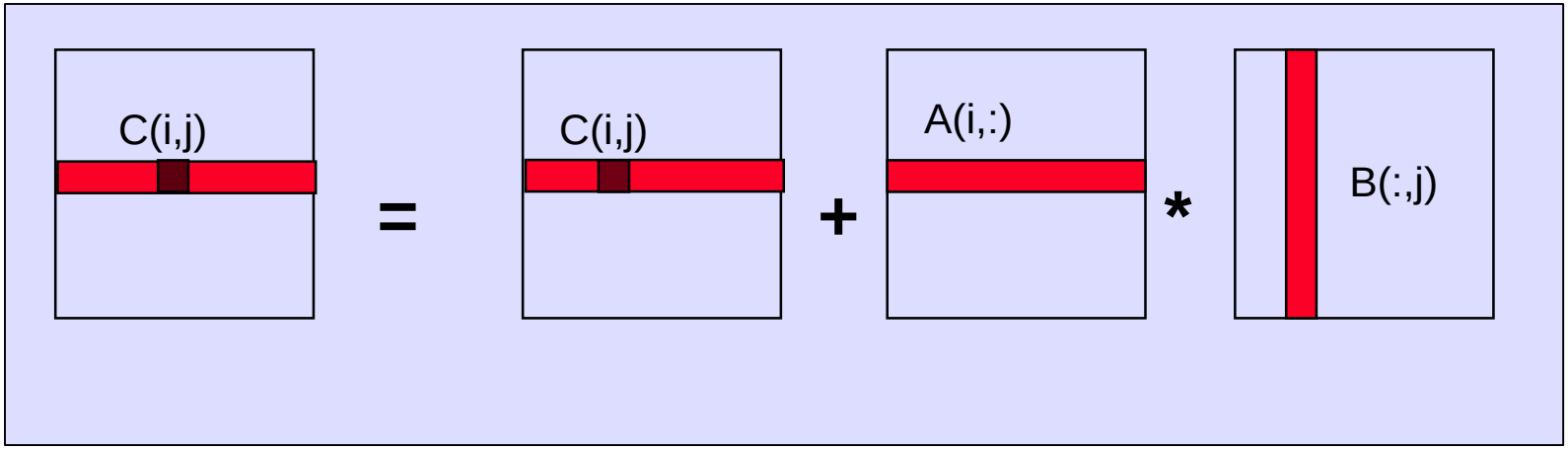
Local memory is consistent between work-items at a barrier.

Global memory is consistent within a work-group at a barrier, but not guaranteed across different work-groups

Consistency of memory shared between **commands** (e.g. kernel invocations) are enforced through **synchronization** (barriers, events, in-order queue)

Optimizing matrix multiplication

There may be significant overhead to manage work-items and work-groups.
So let's have each work-item compute a full row of C



Dot product of a row of A and a column of B for each element of C

Optimizing matrix multiplication

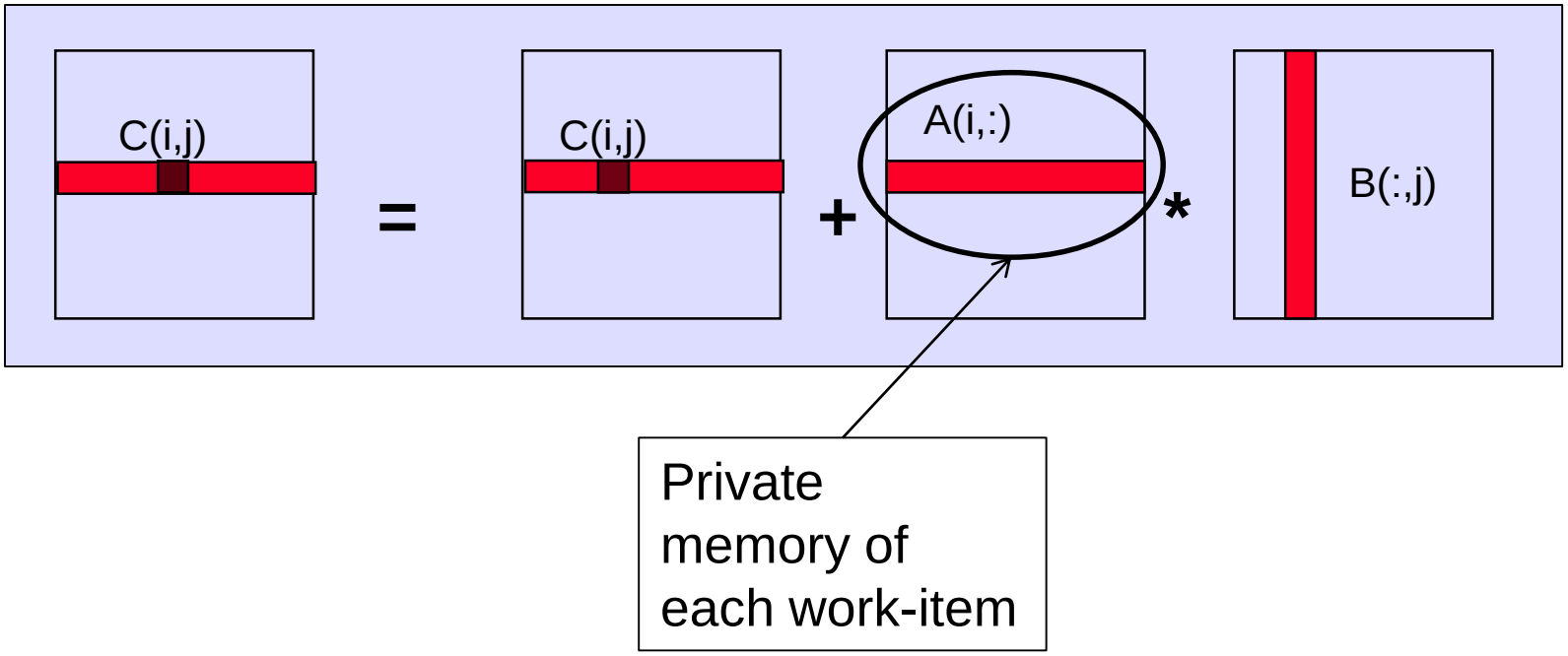
```
__kernel void mmul(const int Mdim, const int Ndim, const int Pdim,
                   __global float* A, __global float* B, __global float* C)
{
    int k, j;
    int i = get_global_id(0);
    float tmp;
    for (j=0; j<Mdim; j++) { // Mdim is width of rows in C
        tmp = 0.0f;
        for (k=0; k<Pdim; k++)
            tmp += A[i*Ndim+k] * B[k*Pdim+j];
        C[i*Ndim+j] += tmp;
    }
}
```

CPU : ~3379 MFLOPS

GPU : ~4195 MFLOPS

Optimizing matrix multiplication

Notice that each element of C in a row uses the same row of A.
Let's copy that row of A into private memory of the work-item that's (exclusively) using it to avoid the overhead of loading it from global memory for each C(i,j) computation.



Optimizing matrix multiplication

```
__kernel void mmul(const int Mdim,  const int Ndim,  const int Pdim, __global float* A,
                   __global float* B,  __global float* C)
{
    int k,j;
    int i = get_global_id(0);
    float Awrk[1024];
    float tmp;
for (k=0; k<Pdim; k++)
        Awrk[k] = A[i*Ndim+k];
      for (j=0; j<Mdim; j++)
      {
        tmp = 0.0f;
        for (k=0; k<Pdim; k++)
          tmp += Awrk[k] * B[k*Pdim+j];
        C[i*Ndim+j] += tmp;
      }
}
```

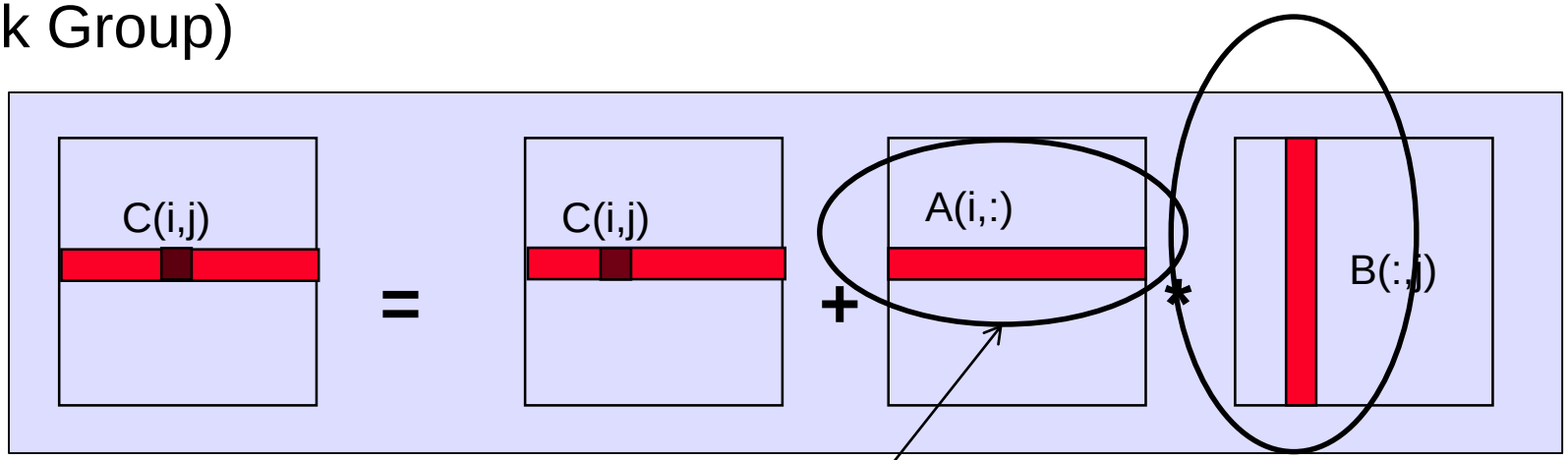
Setup a work array for A in private memory and copy into it from global memory before we start with the matrix multiplications.

CPU : ~3385 MFLOPS

GPU : ~8584 MFLOPS

Optimizing matrix multiplication

We already noticed that each element of C uses the same row of A.
Each work-item in a work-group also uses the same columns of B
So let's store the B columns in *local* memory (shared by Work Items in a Work Group)



Private memory of each work-item

Local memory for each work-group

Optimizing matrix multiplication

Row of C per work-item, A row private, B columns local

```
__kernel void mmul(
    const int Mdim,
    const int Ndim,
    const int Pdim,
    __global float* A,
    __global float* B,
    __global float* C,
    __local float* Bwrk)
{
    int k,j;
    int i = get_global_id(0);
    int iloc = get_local_id(0);
    int nloc = get_local_size(0);
    float Awrk[1024];
    float tmp;
```

```
for (k=0; k<Pdim; k++)
    Awrk[k] = A[i*Ndim+k];
for (j=0; j<Mdim; j++){
    for (k=iloc; k<Pdim; k=k+nloc)
        Bwrk[k] = B[k*Pdim+j];
    barrier(CLK_LOCAL_MEM_FENCE);
    tmp = 0.0f;
    for (k=0; k<Pdim; k++)
        tmp += Awrk[k] * Bwrk[k];
    C[i*Ndim+j] += tmp;
}
```

Pass in a pointer to local memory. Work-items in a group start by copying the columns of B they need into the local memory.

CPU : 10047 MFLOPS
GPU : 8181 MFLOPS

Optimizing Matrix Multiplication

C rows per work item, memory all global
C rows per work item, A private memory
C rows per work item, A private memory, B local memory
Block approach:

- The number of work items must be a multiple of the vector width.
- optimize data reuse using register blocking.
  - Decompose matrices into tiles so that several tiles fit in the private memory
  - Copy tiles into local memory.
  - Do the multiplication over the tiles

  CPU : ~1500 MFLOPS
  GPU : ~200,000 MFLOPS

Optimization Issues

Efficient access to memory

- Memory coalescing
  - Ideally get work item *i* to access *data[i]* and work item *j* to access *data[j]* at the same time etc.
- Memory alignment
  - Padding arrays to keep everything aligned to multiples of 16 or 32 bytes

Number of work items and work group sizes

- Ideally want at least 4 work items per PE in a Compute Unit on GPUs
- More is better, but diminishing returns, and there is an upper limit
  - Each work item consumes PE finite resources (registers etc)

Work item divergence

- What happens when work items branch?
- Actually a SIMD data parallel model
- Both paths (if-else) may need to be executed, so avoid where possible

Portability of OpenCL

Don’t optimize too hard for any one platform:

- Don’t write specifically for certain vector width sizes.
- Be careful not to max out specific sizes of local/global memory.
- OpenCL’s vector data types have varying degrees of support – faster on some devices, slower on others.
- Some devices have caches in their memory hierarchies, some don’t, and it can make a big difference to your performance without you realizing.
- Choosing the allocation of Work-Items to Work-Groups and dimensions on your kernel launches.
- Performance differences between unified vs. disjoint host/global memories.
- Double precision performance varies considerably from device to device.

Performance

Most important to keep the fastest devices busy.

- Less important if slower devices finish slightly earlier than faster ones.

Be careful to avoid using the CPU for both OpenCL host code and OpenCL device code at the same time.

Assigning Work-Items to Work-Groups will need different treatment for different devices.

- For example, CPUs tend to prefer 1 Work-Item per Work-Group, while GPUs prefer lots of Work-Items per Work-Group (usually a multiple of the number of PEs per compute unit, i.e. 32, 64 etc).

The OpenCL run-time will have a go at choosing good EnqueueNDRangeKernel dimensions for you.

Write adaptive code that makes decisions at run-time.

Resources

https://www.khronos.org/opencl/

http://www.khronos.org/opencl/resources

http://www.khronos.org/files/opencl-quick-reference-card.pdf

https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf

https://www.khronos.org/files/opencl20-quick-reference-card.pdf