Lecture 17

Multiple GPUs, Double Buffering, Local Memory

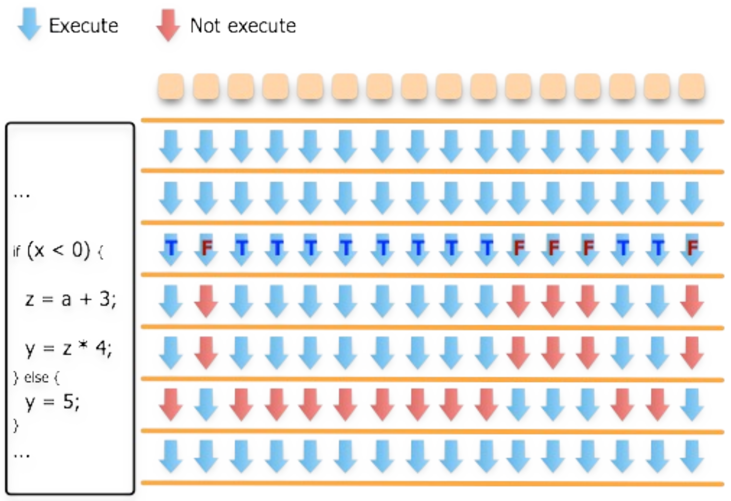
이재진
서울대학교 데이터사이언스대학원
서울대학교 공과대학 컴퓨터공학부
http://aces.snu.ac.kr/~jlee

Hardware Scheduling Units in GPUs

- Basic unit of GPUs for scheduling
  - All threads in it processes a single instruction at the same time in SIMD fashion
  - Lock-step
- NVIDIA - warp
  - 32 hardware threads (work-items)
- AMD - wavefront
  - 64 hardware threads (work-items)
- Other vendors may have different names

Divergence

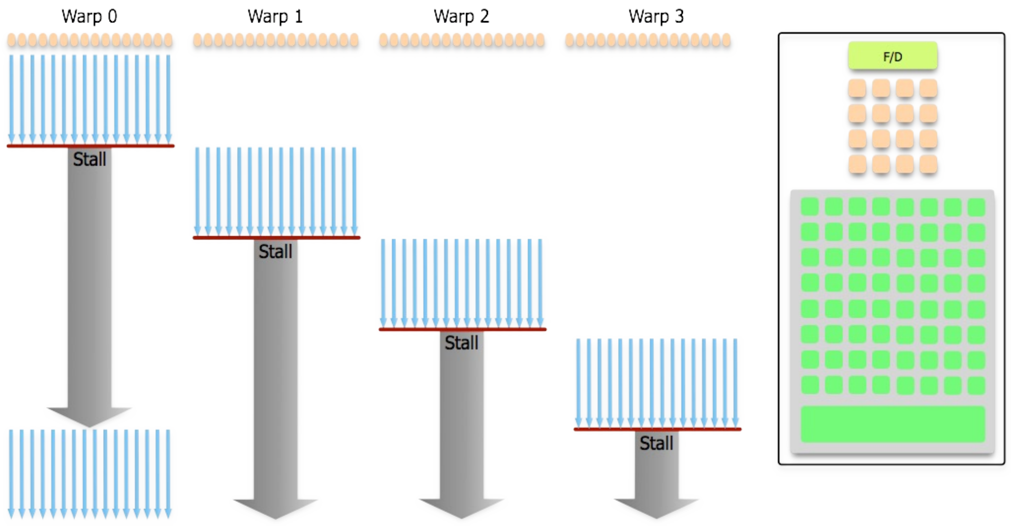
- When work-items in the same work-group follow different paths of control flow, they diverge in their execution
  - If - then - else
  - Loops with different loop bounds for different work-items
- Low degree of divergence will be better
- Pick a work-group size that is a multiple of the warp size



```
...
if (x < 0) {
  z = a + 3;
  y = z * 4;
} else {
  y = 5;
}
...
```

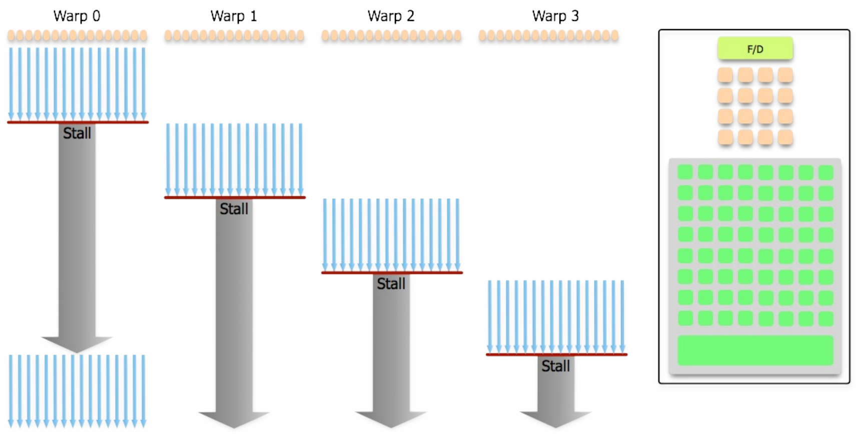
Occupancy

- The number of active warps per Streaming Multiprocessor (AMD calls it a Streaming Core)
  - Computed at compile time
- It describes how well the resources of the SM are being utilized



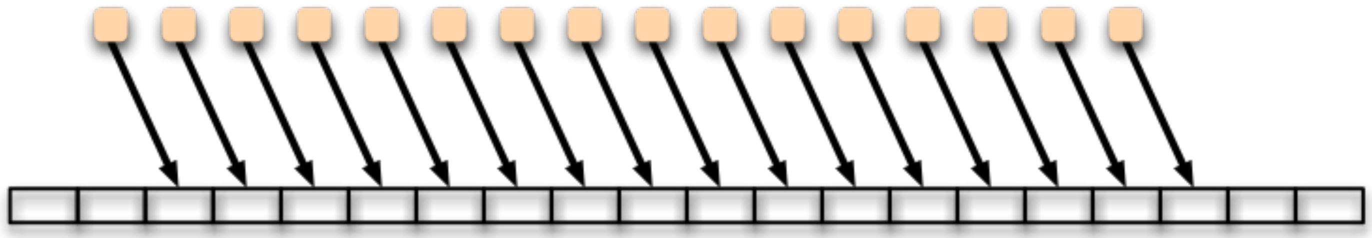
Occupancy (cont'd)

- The maximum number of registers required by a kernel must be available for all threads in a warp
- The maximum size of local memory required by a kernel must be available for all threads in a warp
- The maximum number of active threads and warps per SM is limited
- Consider the above three factors



Memory Coalescing

- The same instruction for all work-items in a warp accesses consecutive global memory locations
  - The hardware coalesces all of these accesses to a consolidated access
  - To achieve the peak global memory bandwidth
- For example, work-item 0 accesses global memory location N, work-item 1 accesses $N + 1$, etc.



Thread Granularity

- Put more work into each work-item and use fewer work-items
  - May reduce the kernel launching overhead
  - May remove redundant computations between work-items
  - May increase the number of registers resulting in low occupancy
  - May reduce the number of work-groups resulting in making the SM underutilized

Host-device Data Transfer

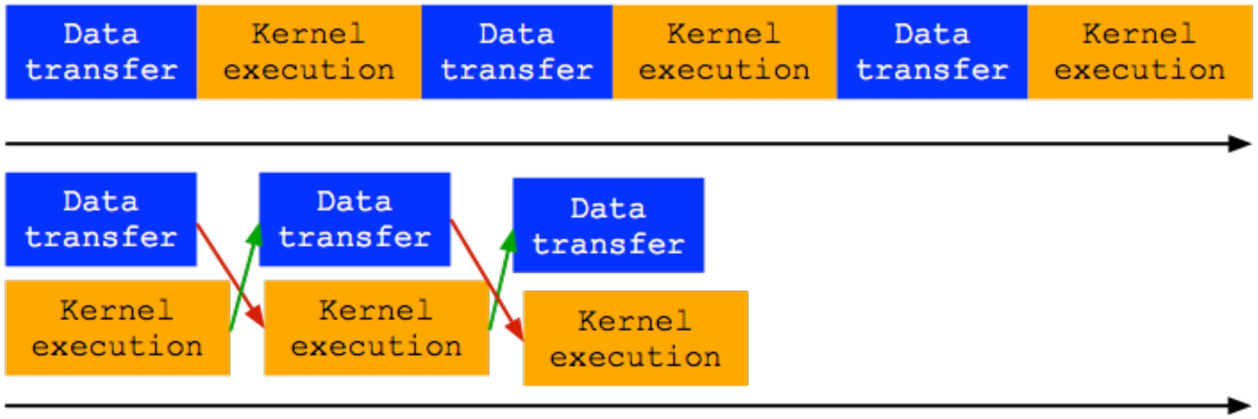
- Data transfer between the host and the device has much lower bandwidth than global memory accesses
  - PCI-E : a few GB/s
  - Global memory: a few hendred GB/s

- Minimize data transfer between the host and the device
  - Better to recompute on the accelerator
  - Use the global memory on the accelerator for intermediate data

- One large transfer is much faster than many small ones

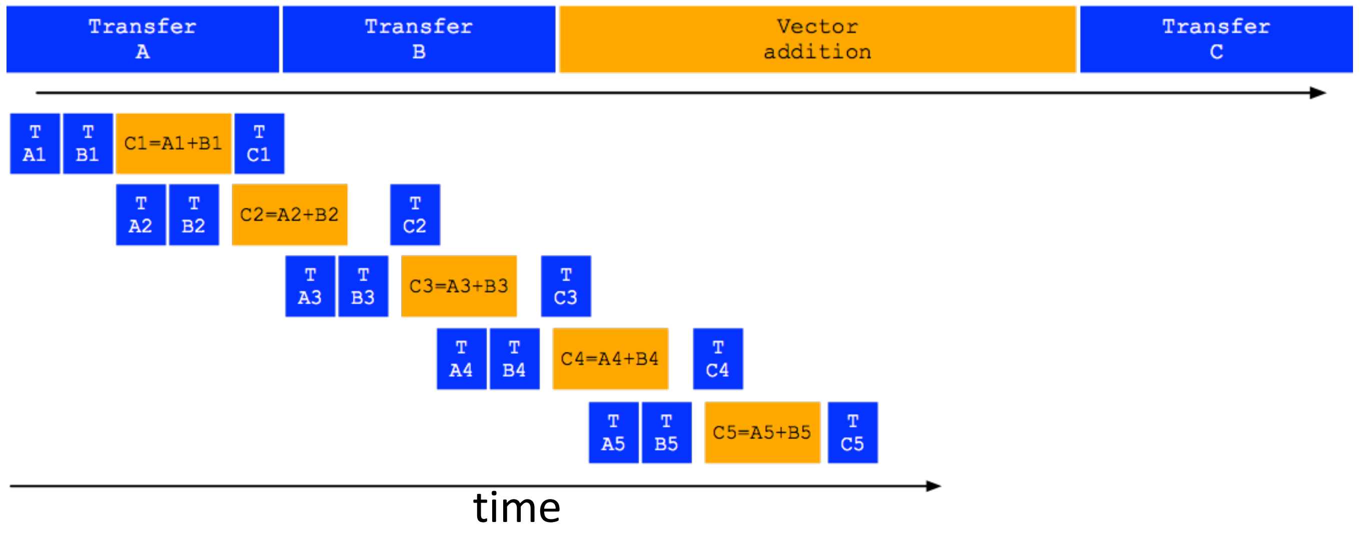
Asynchronous Copy

- Overlap of data transfer and code execution
    - Simultaneously execute a kernel while performing a copy between device and host memory
- Relevant only for accelerators using PCI-E bus
- Use two queues
    - One for data transfer and one for execution
    - Use events to enforce dependences



Pipelining (Double Buffering)

- Vector addition example
  - A + B = C
  - Divide large vectors into segments
    - A1, A2, …, An
    - B1, B2, …, Bn
    - C1, C2, …, Cn
  - Overlap transfer and addition



Using Local Memory

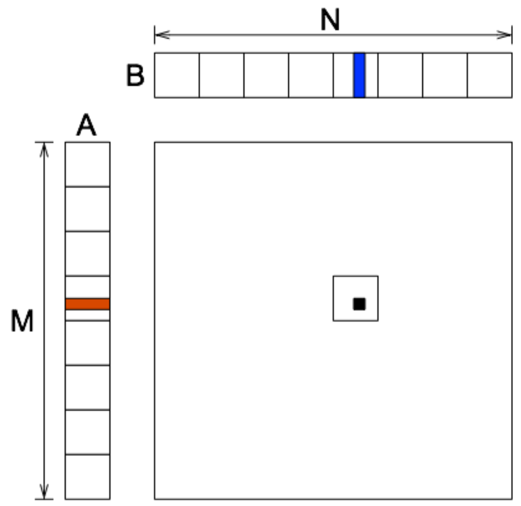
- Local memory is local to a work-group
  - Shared by all work-items of work-group
  - Used to cache global memory
  - Low latency access
- Two ways to allocate it
  - Statically, inside the kernel
  - Dynamically, from the host as a parameter

Allocating Local Memory

- Inside a kernel
  - Declare local memory as a static array
  - Use the keyword `__local`
- Kernel with parameter in local memory
  - Allocation done during set of kernel arguments

Matrix Multiplication Using Local Memory

- Every work-item takes care of one element in C



```
__kernel void MatMul(__global float* a,
                     __global float* b,
                     __global float* c,
                     int N)
{
    int row = get_global_id(1);
    int col= get_global_id(0);
    float sum = 0.0f;
    for (int i= 0; i < T_SIZE; i++) {
        sum += a[row*T_SIZE+i]
                 * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

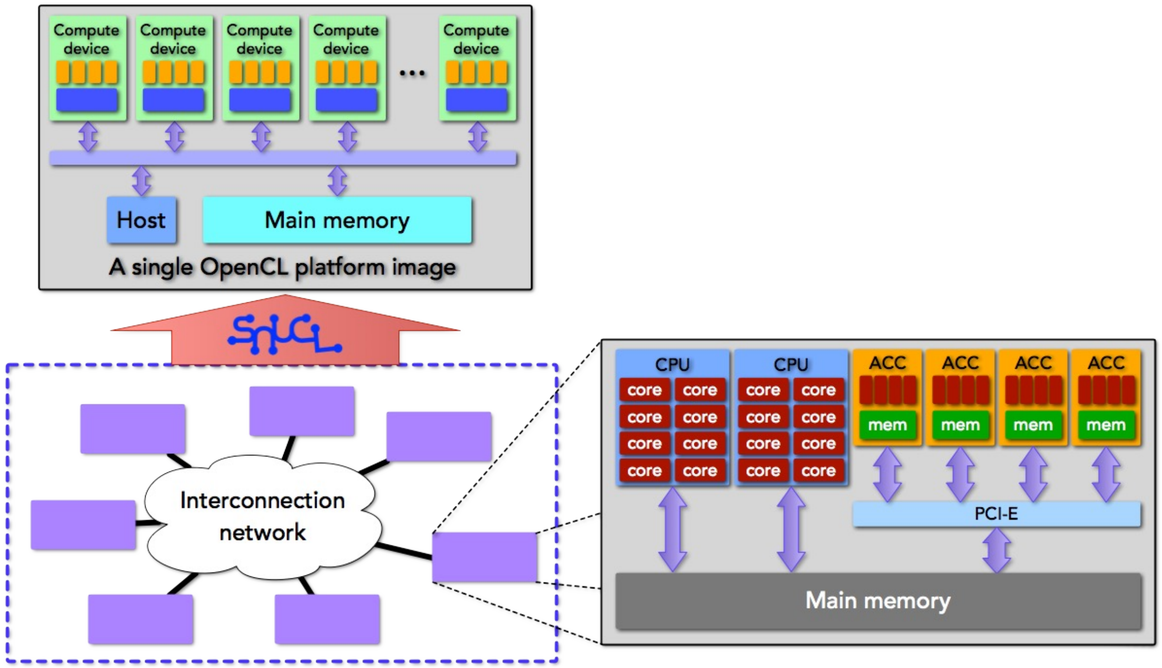
Matrix Multiplication Using Local Memory (cont'd)

```
__kernel void TMatMul(__global float* a,
                      __global float* b,
                      __global float* c,intN,
                      __local float tile[T_SIZE][T_SIZE])
{
    int row = get_global_id(1);
    int col= get_global_id(0);
    float sum = 0.0f;
    int x = get_local_id(0);
    int y = get_local_id(1);

    tile[y][x] = a[row*T_SIZE+x];
    for (int i= 0; i< T_SIZE; i++) {
        sum += tile[y][i]* b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

Illusion of a Single OpenCL Platform Image

- If the programmer can write applications for heterogeneous clusters using only OpenCL
  - Easy to program
  - More portable program

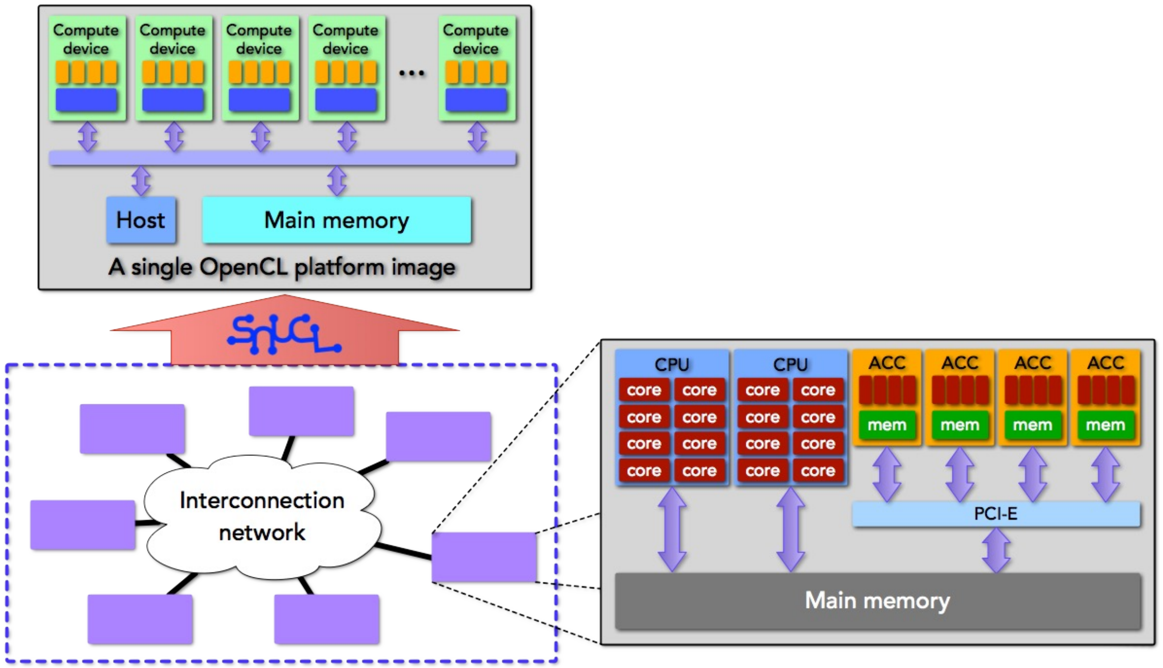


SnuCL

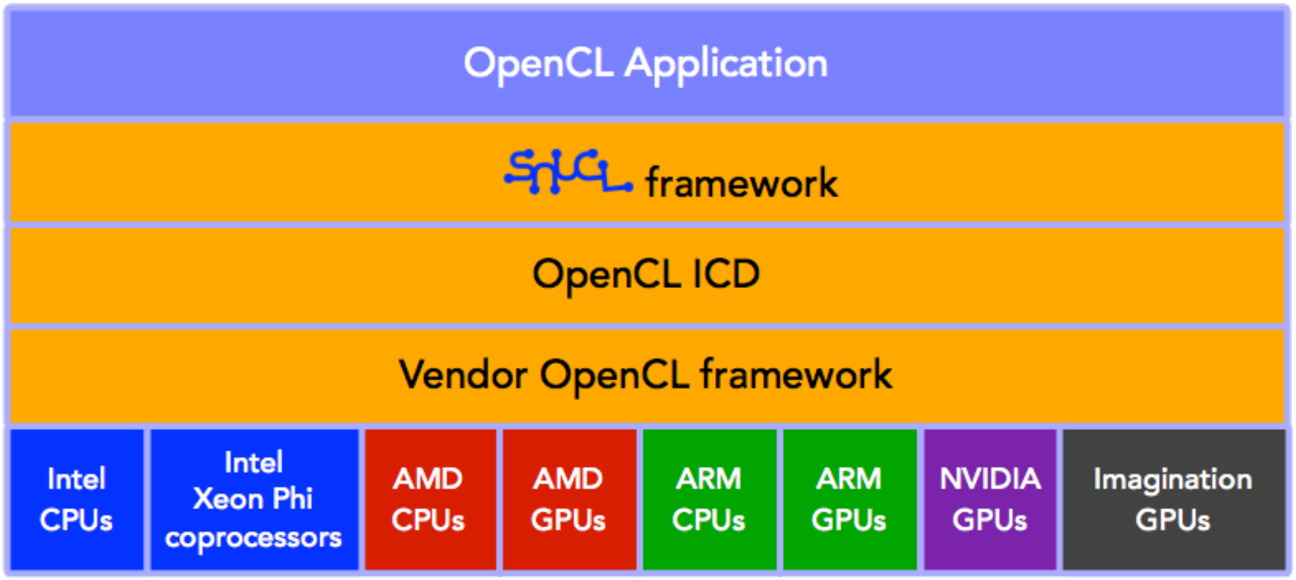
- An OpenCL framework
  - Platform layer + runtime + kernel compiler
- Freely available, open-source software developed at Seoul National University
  - http://aces.snu.ac.kr
  - Supports OpenCL 1.2
  - Passed most of OpenCL conformance tests
- Supports x86 CPUs, ARM CPUs, AMD GPUs, NVIDIA GPUs, Intel Xeon Phi coprocessors (from July, 2013)
- With SnuCL, an OpenCL application written for a single operating system instance runs on a heterogeneous cluster without any modification

How to Achieve the Illusion?

- SnuCL runtime provides the illusion
  - Handles communication between nodes
  - Efficient buffer and consistency management



Using ICD in SnuCL



SnuCL's Approach

- Exploits the OpenCL ICD
- However,
  - No need to explicitly specify a specific framework
  - Can share objects (buffers, events, etc.) between different frameworks in the same application
- Works for heterogeneous clusters, too

SnuCL's Approach (cont'd)

- Naturally extends the original OpenCL semantics to the heterogeneous cluster environment
  - Provides an illusion of a heterogeneous system running a single OS instance

- With SnuCL, an OpenCL application written for a single OS instance runs on a heterogeneous cluster without any modification

The Effect of Using SnuCL

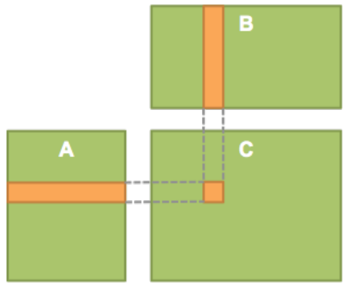
- Copy buffers between different nodes in the cluster environment (Buffer A → Buffer B)

Previous approach (Mixture of MPI and OpenCL)	SnuCL (OpenCL only)

MPI_Init(..);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
…
cl_mem bufferA = clCreateBuffer(…);
cl_mem bufferB = clCreateBuffer(…);
…
void *temp = malloc(…);
if (rank == SRC_DEV) {
  clEnqueueReadBuffer(cq, bufferA, …, temp, …);
  MPI_Send(temp, …, DST_DEV, …);
} else if (rank == DST_DEV) {
  MPI_Recv(temp, …, SRC_DEV, …);
  clEnqueueWriteBuffer(cq, bufferB, …, temp, …);
}
…
MPI_Finalize();
	…
	cl_mem bufferA = clCreateBuffer(…);
	cl_mem bufferB = clCreateBuffer(…);
	…
	clEnqueueCopyBuffer(cq, bufferA, bufferB, …);
	…

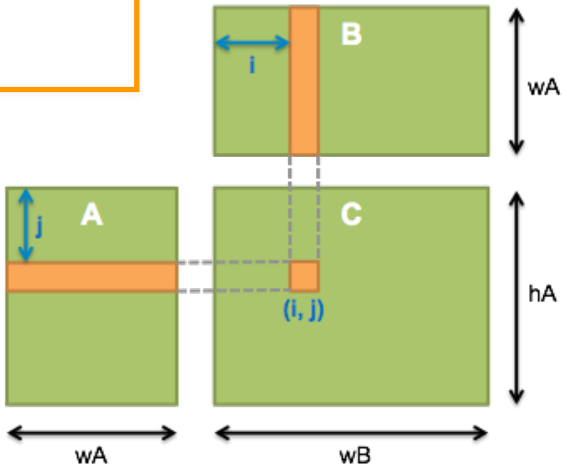
Matrix Multiplication Example

- Each work-item computes an element of C
  - Loads a row of A
  - Loads a column of B
  - Computes the dot product
- Five different implementations
  - For a single GPU under a single OS instance
  - For multiple GPUs under a single OS instance
  - For multiple GPUs in the cluster using,
    - OpenCL + MPI
    - SnuCL
    - SnuCL with collective communication extensions



For a Single GPU

```
__kernel void matrixmul(__global float* C, __global float* A,
                        __global float* B, int wA, int wB)
{    int i = get_global_id(0);
     int j = get_global_id(1);
     int k;
     float acc = 0.0;
     for (k = 0; k < wA; k++)
         acc += A[j * wA + k] * B[k * wB + i];
    C[j * wB + i] = acc;
}
```



The Host Program for a Single GPU

```
#include <CL/cl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE  1024
const char* kernel_src = "__kernel void matrixmul(__global float* C, "
                         "  __global float* A, __global float* B, "
                         "  int wA, int wB) {"
                         "  int i = get_global_id(0);"
                         "  int j = get_global_id(1);"
                         "  int k;"
                         "  float acc = 0.0;"
                         "  for (k = 0; k < wA; k++) {"
                         "    acc += A[j * wA + k] * B[k * wB + i];"
                         "  }"
                         "  C[j * wB + i] = acc;"
                         "}";
```

The Host Program for a Single GPU (cont'd)

```
int main(int argc, char** argv) {
  cl_platform_id    platform;
  cl_device_id      device;
  cl_context        context;
  cl_command_queue  command_queue;
  cl_program        program;
  cl_kernel         kernel;
  cl_mem            bufferA;
  cl_mem            bufferB;
  cl_mem            bufferC;
  float*            hostA;
  float*            hostB;
  float*            hostC;
  int               wA, hA, wB, hB, wC, hC;
  size_t            sizeA, sizeB, sizeC;
```

The Host Program for a Single GPU (cont'd)

```
  wA = hA = wB = SIZE;
  hB = wA;
  wC = wB;
  hC = hA;
  sizeA = wA * hA * sizeof(float);
  sizeB = wB * hB * sizeof(float);
  sizeC = wC * hC * sizeof(float);
  hostA = (float*) malloc(sizeA);
  hostB = (float*) malloc(sizeB);
  hostC = (float*) malloc(sizeC);

// Initialize hostA and hostB
  ...
```

The Host Program for a Single GPU (cont'd)

```
clGetPlatformIDs(1, &platform, NULL);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
context = clCreateContext(0, 1, &device, NULL, NULL, NULL);
command_queue = clCreateCommandQueue(context, device, 0, NULL);
bufferA = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeA, NULL, NULL);
bufferB = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeB, NULL, NULL);
bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeC, NULL, NULL);
size_t kernel_src_len = strlen(kernel_src);
program = clCreateProgramWithSource(context, 1,
                                    (const char**) &kernel_src,
                                    &kernel_src_len, NULL);
clBuildProgram(program, 1, &device, NULL, NULL, NULL);
kernel = clCreateKernel(program, "matrixmul", NULL);
```

The Host Program for a Single GPU (cont'd)

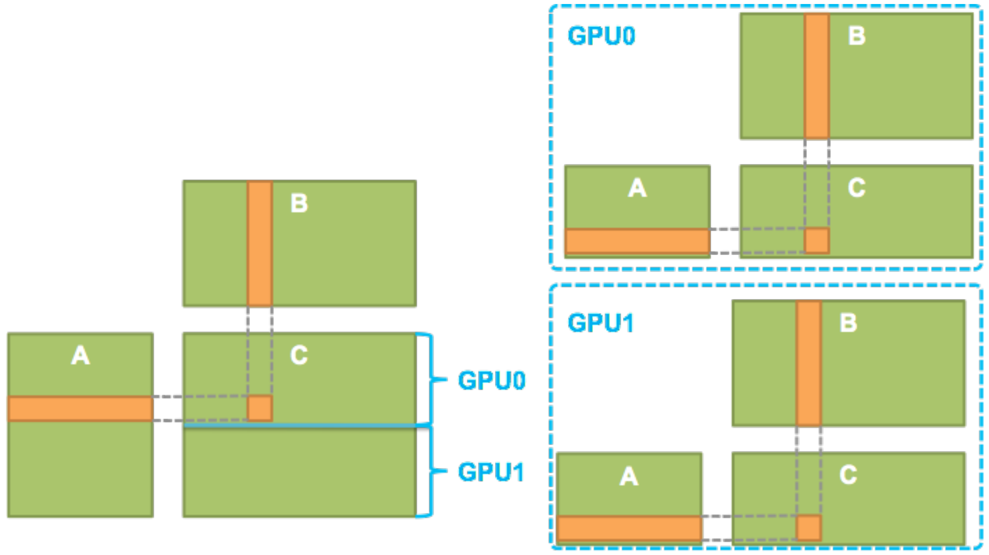
```
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*) &bufferC);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*) &bufferA);
clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*) &bufferB);
clSetKernelArg(kernel, 3, sizeof(cl_int), (void*) &wA);
clSetKernelArg(kernel, 4, sizeof(cl_int), (void*) &wB);
clEnqueueWriteBuffer(command_queue, bufferA, CL_FALSE, 0,
                     sizeA, hostA, 0, NULL, NULL);
clEnqueueWriteBuffer(command_queue, bufferB, CL_FALSE, 0,
                     sizeB, hostB, 0, NULL, NULL);
```

The Host Program for a Single GPU (cont'd)

```
  size_t global[2] = { wC, hC };
  size_t local[2] = { 16, 16 };
  clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
                         global, local, 0, NULL, NULL);
  clEnqueueReadBuffer(command_queue, bufferC, CL_TRUE, 0,
                      sizeC, hostC, 0, NULL, NULL);
// Print hostC
 ...  return 0;
}
```

Matrix Multiplication for Multiple GPUs

- Partition the kernel index space into N regions for N GPUs
  - Partition along the y-axis



The Host Program for Multiple GPUs

- Assume that the system contains 4 GPUs

```
...
#include <string.h>
#define SIZE  1024
#define MAX_DEV 4
const char* kernel_src = "__kernel void matrixmul(__global float* C, "
                         ...
                         "}";
```

The Host Program for Multiple GPUs (cont'd)

```
int main(int argc, char** argv) {
  cl_platform_id    platform;
  cl_device_id      device[MAX_DEV];
  cl_context        context;
  cl_command_queue  command_queue[MAX_DEV];
  cl_program        program;
  cl_kernel         kernel[MAX_DEV];
  cl_mem            bufferA[MAX_DEV];
  cl_mem            bufferB[MAX_DEV];
  cl_mem            bufferC[MAX_DEV];
  float*            hostA;
  float*            hostB;
  float             hostC;
  int               wA, hA, wB, hB, wC, hC;
  size_t            sizeA, sizeB, sizeC;
  int               ndev;
```

The Host Program for Multiple GPUs (contd.)

```
wA = hA = wB = SIZE;
hB = wA;
wC = wB;
hC = hA;
sizeA = wA * hA * sizeof(float);
sizeB = wB * hB * sizeof(float);
sizeC = wC * hC * sizeof(float);
hostA = (float*) malloc(sizeA);
hostB = (float*) malloc(sizeB);
hostC = (float*) malloc(sizeC);

// Initialize hostA and hostB
...
```

The Host Program for Multiple GPUs (cont'd)

```
clGetPlatformIDs(1, &platform, NULL);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 0, NULL,
               (unsigned int*) &ndev);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, ndev, device, NULL);
context = clCreateContext(0, ndev, device, NULL, NULL, NULL);

for (i = 0; i < ndev; i++)
  command_queue = clCreateCommandQueue(context, device[i], 0, NULL);
for (i = 0; i < ndev; i++) {
  bufferA[i] = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              sizeA/ndev, NULL, NULL);
  bufferB[i] = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              sizeB, NULL, NULL);
  bufferC[i] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                              sizeC/ndev, NULL, NULL);
}
```

The Host Program for Multiple GPUs (cont'd)

```
size_t kernel_src_len = strlen(kernel_src);
program = clCreateProgramWithSource(context, 1,
                                    (const char**) &kernel_src,
                                    &kernel_src_len, NULL);
clBuildProgram(program, ndev, device, NULL, NULL, NULL);
for (i = 0; i < ndev; i++) {
  kernel[i] = clCreateKernel(program, "matrixmul", NULL);
}
```

The Host Program for Multiple GPUs (cont'd)

```
for (i = 0; i < ndev; i++) {
  clSetKernelArg(kernel[i], 0, sizeof(cl_mem), (void*) &bufferC[i]);
  clSetKernelArg(kernel[i], 1, sizeof(cl_mem), (void*) &bufferA[i]);
  clSetKernelArg(kernel[i], 2, sizeof(cl_mem), (void*) &bufferB[i]);
  clSetKernelArg(kernel[i], 3, sizeof(cl_int), (void*) &wA);
  clSetKernelArg(kernel[i], 4, sizeof(cl_int), (void*) &wB);
}
for (i = 0; i < ndev; i++) {
  clEnqueueWriteBuffer(command_queue[i], bufferA[i], CL_FALSE, 0,
                       sizeA/ndev,
                       (void*) ((size_t) hostA + (sizeA/ndev)*i),
                       0, NULL, NULL);
  clEnqueueWriteBuffer(command_queue[i], bufferB[i], CL_FALSE, 0,
                       sizeB, hostB, 0, NULL, NULL);
}
```

The Host Program for Multiple GPUs (cont'd)

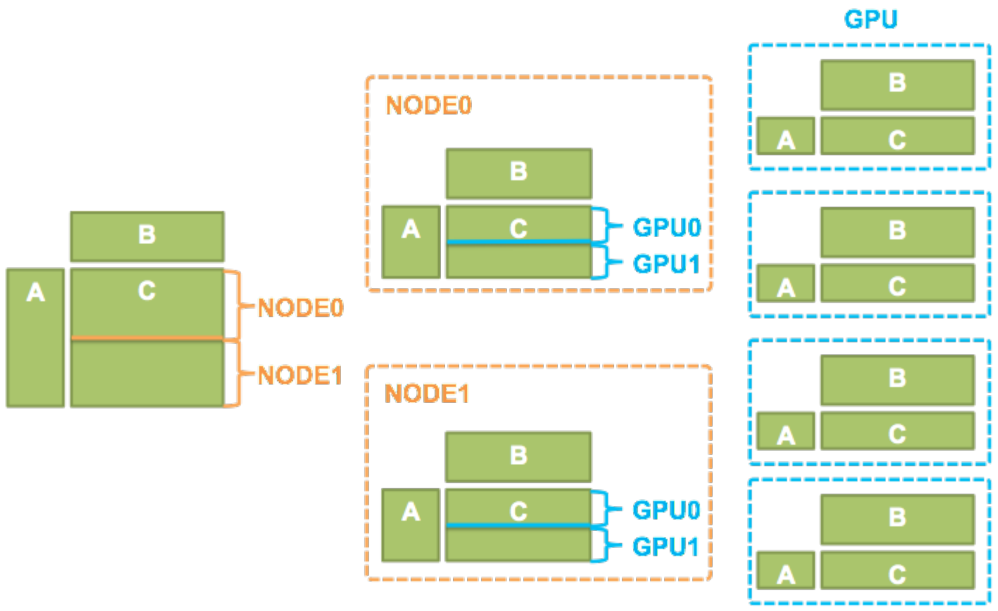
```
  size_t global[2] = { wC, hC/ndev };
  size_t local[2] = { 16, 16 };

  for (i = 0; i < ndev; i++)
    clEnqueueNDRangeKernel(command_queue[i], kernel[i], 2, NULL,
                           global, local, 0, NULL, NULL);

  for (i = 0; i < ndev; i++)
     clEnqueueReadBuffer(command_queue[i], bufferC[i], CL_TRUE, 0,
                         sizeC/ndev,
                         (void*) ((size_t) hostC+(sizeC/ndev)*i),
                         0, NULL, NULL);
// Print hostC
 ...
  return 0;
}
```

Matrix Multiplication for the Cluster

- Hierarchical Partitioning



Write Buffers in MPI+OpenCL

```
if (rank == 0) {
  for (i = 1; i < nnode; i++) {
    MPI_Isend((void*) ((size_t) hostA+(sizeA/nnode) * i),
               (int) sizeA / nnode, MPI_CHAR, i, 0, MPI_COMM_WORLD, &request[0]);
    MPI_Isend(hostB, (int) sizeB, MPI_CHAR, i, 0, MPI_COMM_WORLD, &request[1]);
  }
} else {
  MPI_Irecv(hostA, (int) sizeA / nnode, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &request[0]);
  MPI_Irecv(hostB, (int) sizeB, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &request[1]);
}
MPI_Waitall(2, request, status);

for (i = 0; i < ndev; i++) {
  clEnqueueWriteBuffer(command_queue[i], bufferA[i], CL_FALSE, 0,
                       sizeA/nnode/ndev,
                       (void*) ((size_t) hostA+(sizeA/nnode/ndev)*i),
                       0, NULL, NULL);
  clEnqueueWriteBuffer(command_queue[i], bufferB[i], CL_FALSE, 0,
                       sizeB, B, 0, NULL, NULL);
}
```

Read Buffers in MPI+OpenCL

```
for (i = 0; i < ndev; i++) {
  clEnqueueReadBuffer(command_queue[i], bufferC[i], CL_TRUE, 0,
                      sizeC/nnode/ndev,
                      (void*) ((size_t) hostC + (sizeC/nnode/ndev) * i),
                      0, NULL, NULL);
}
if (rank == 0) {
  for (i = 1; i < nnode; i++) {
    MPI_Irecv((void*) ((size_t) hostC+(sizeC/nnode)*i),
                       (int) sizeC/nnode,MPI_CHAR,
                       i, 0, MPI_COMM_WORLD, &request[0]);
  }
} else {
  MPI_Isend(host C, (int) sizeC/nnode, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &request[0]);
}
MPI_Wait(request, status);
```

SnuCL for the Cluster

- Assume that the cluster contains 32 GPUs
- Modify **`#define MAX_DEV 4`** in the OpenCL host program for multiple GPUs to **`#define MAX_DEV 32`**

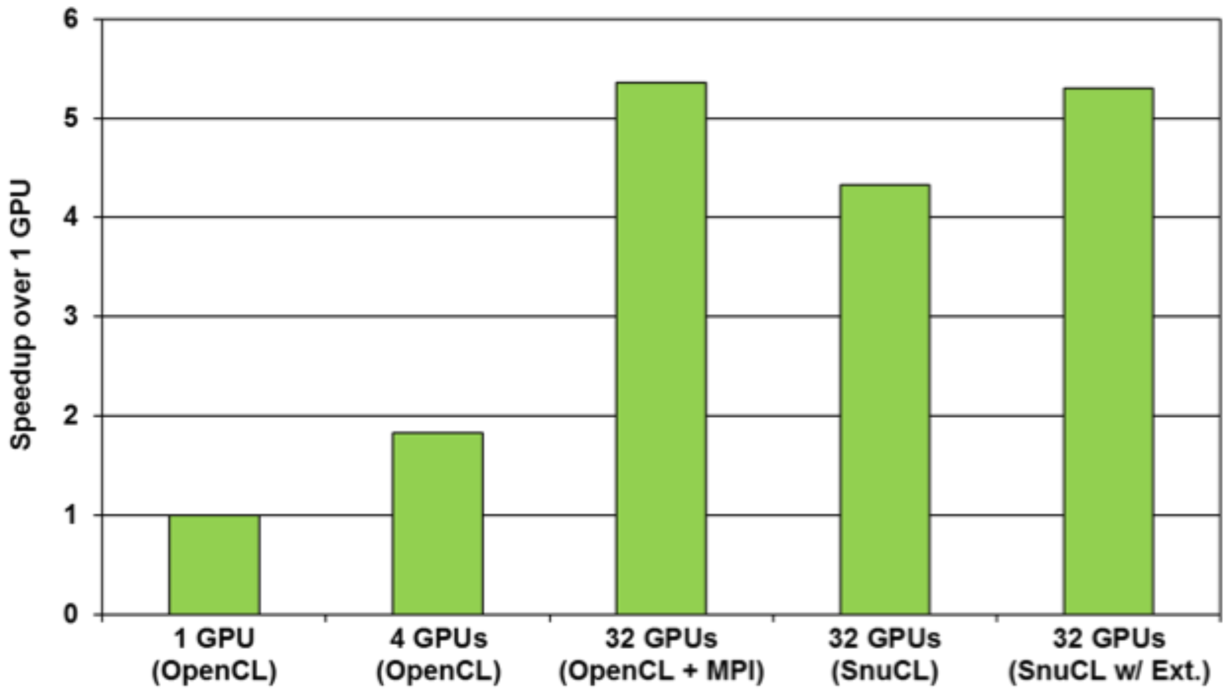
SnuCL Extensions for Multiple GPUs in the Cluster

```
for (i = 0; i < ndev; i++) {
  clEnqueueWriteBuffer(command_queue[i], bufferA[i], CL_FALSE, 0, sizeA/ndev,
                       (void*) ((size_t) hostA+(sizeA/ndev)*i),
                       0, NULL, NULL);
}
for (i = 0; i < ndev; i++) {
  clEnqueueWriteBuffer(command_queue[i], bufferB[i], CL_FALSE, 0, sizeB,
                       hostB, 0, NULL, NULL);
}
```

$\downarrow$

```
clEnqueueWriteBuffer(command_queue[0], bufferB[0], CL_TRUE, 0, sizeB,
                     hostB, 0, NULL, NULL);
clEnqueueBroadcastBuffer(cmq + 1, bufferB[0], ndev-1, bufferB+1,
                         0, NULL, sizeB, 0, NULL, NULL);
```

Matrix Multiplication Performance


Speedup over 1 GPU
1 GPU (OpenCL)
4 GPUs (OpenCL)
32 GPUs (OpenCL + MPI)
32 GPUs (SnuCL)
32 GPUs (SnuCL w/ Ext.)