Optimizing Performance

A Collaboration Between
David Kaeli, Northeastern University
Benedict R. Gaster, AMD

Instructor Notes

- This lecture discusses three important optimizations
- The performance impact of mapping threads to data on the GPU is subtle but extremely important. Examples are shown (including a detailed matrix transpose) along with actual empirical results
- The number of threads that are active on the GPU can also play a large part in achieving good performance, and so the subtleties of GPU *occupancy* are discussed
- Vectorization is particularly important for AMD GPUs and is briefly discussed as well

Topics

- Thread mapping
  - Choosing a proper mapping
  - Optimizing with local memory
- Device occupancy
- Vectorization

Thread Mapping

- Thread mapping determines which threads will access which data
    - Proper mappings can align with hardware and provide large performance benefits
    - Improper mappings can be disastrous to performance
- The paper *Static Memory Access Pattern Analysis on a Massively Parallel GPU* by Jang, et. al focuses on the task of effectively mapping threads to the data access patterns of an algorithm

Thread Mapping

- By using different mappings, the same thread can be assigned to access different data elements
  - The examples below show three different possible mappings of threads to data (assuming the thread id is used to access an element)

Mapping

```
int tid =
get_global_id(1) *
get_global_size(0) +
get_global_id(0);
```

```
int tid =
get_global_id(0) *
get_global_size(1) +
get_global_id(1);
```

```
int group_size =
get_local_size(0) *
get_local_size(1);

int tid =
get_group_id(1) *
get_num_groups(0) *
group_size +
get_group_id(0) *
group_size +
get_local_id(1) *
get_local_size(0) +
get_local_id(0)
```

Thread IDs

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

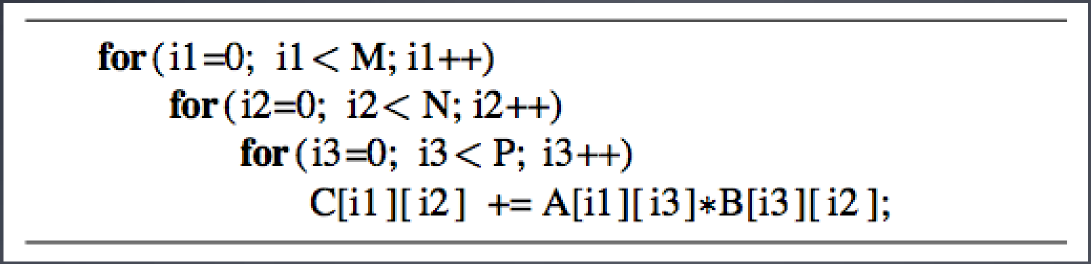
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

*assuming 2x2 groups

Thread Mapping

- Consider a serial matrix multiplication algorithm



```
for (i1=0;  i1 < M; i1++)
    for (i2=0;  i2 < N; i2++)
        for (i3=0;  i3 < P; i3++)
            C[i1][i2]  += A[i1][i3]*B[i3][i2];
```

- This algorithm is suited for output data decomposition
  - We will create *NM* threads
    - Effectively removing the outer two loops
  - Each thread will perform *P* calculations
    - The inner loop will remain as part of the kernel
- Should the index space be MxN or NxM?

Thread Mapping

- Thread mapping 1: with an MxN index space, the kernel would be:

```
int  tx  =  get_global_id(0);
int  ty  =  get_global_id(1);
for(i3=0;  i3<P; i3++)
     C[tx][ty]  += A[tx][i3]*B[i3][ty];
```

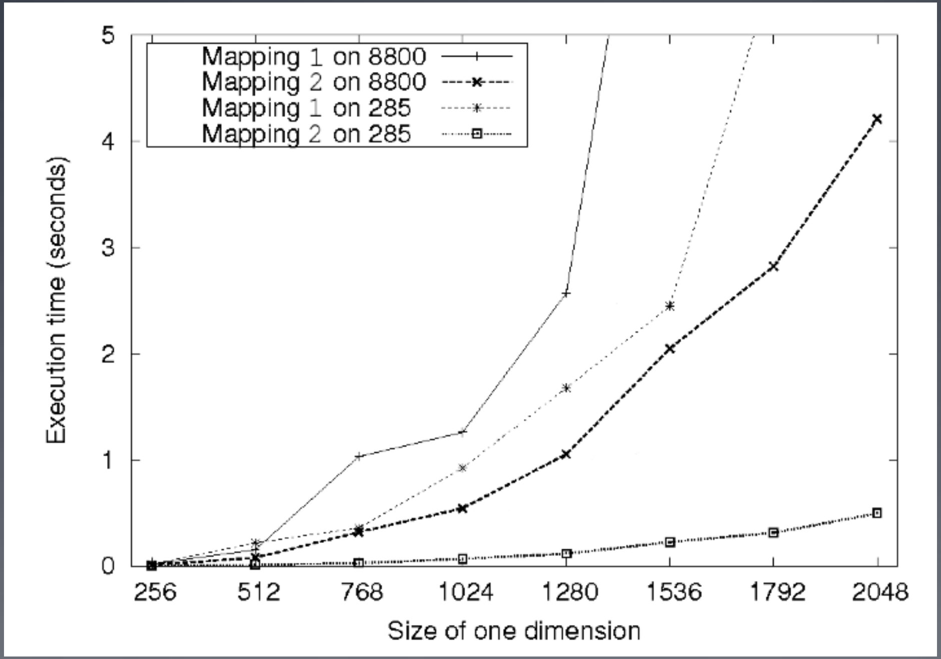
- Thread mapping 2: with an NxM index space, the kernel would be:

```
int  tx  =  get_global_id (0);
int  ty  =  get_global_id (1);
for(i3=0;  i3<P; i3++)
     C[ty][tx]  += A[ty][i3]*B[i3][tx];
```

- Both mappings produce functionally equivalent versions of the program

Thread Mapping

- This figure shows the execution of the two thread mappings on NVIDIA GeForce 285 and 8800 GPUs



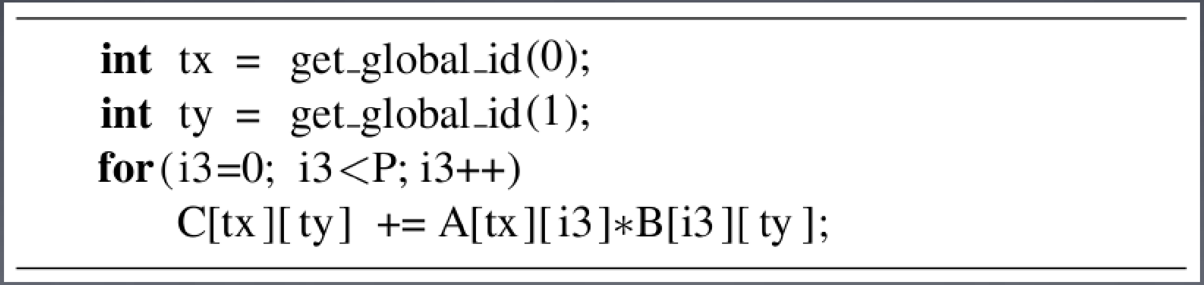
- Notice that mapping 2 is far superior in performance for both GPUs

Thread Mapping

- The discrepancy in execution times between the mappings is due to data accesses on the global memory bus
  - Assuming row-major data, data in a row (i.e., elements in adjacent columns) are stored sequentially in memory
  - To ensure coalesced accesses, consecutive threads in the same wavefront should be mapped to columns (the second dimension) of the matrices
    - This will give coalesced accesses in Matrices B and C
    - For Matrix A, the iterator *i3* determines the access pattern for row-major data, so thread mapping does not affect it

Thread Mapping

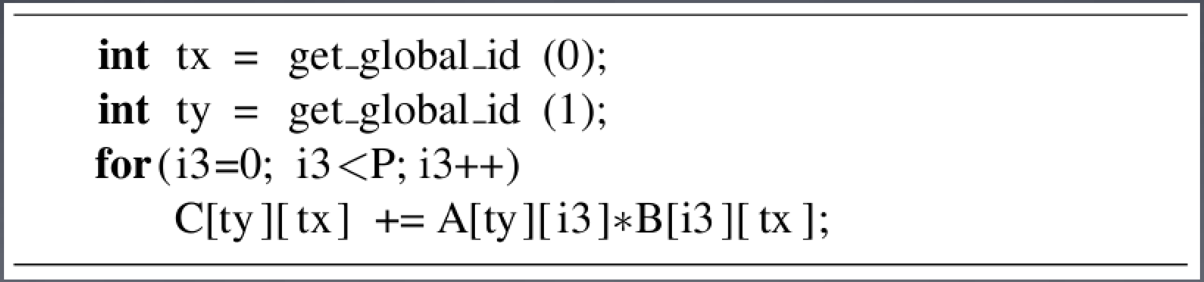
- In mapping 1, consecutive threads (*tx*) are mapped to different rows of Matrix C, and non-consecutive threads (*ty*) are mapped to columns of Matrix B
    - The mapping causes inefficient memory accesses



```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

Thread Mapping

- In mapping 2, consecutive threads (*tx*) are mapped to consecutive elements in Matrices B and C
  - Accesses to both of these matrices will be coalesced
    - Degree of coalescence depends on the workgroup and data sizes



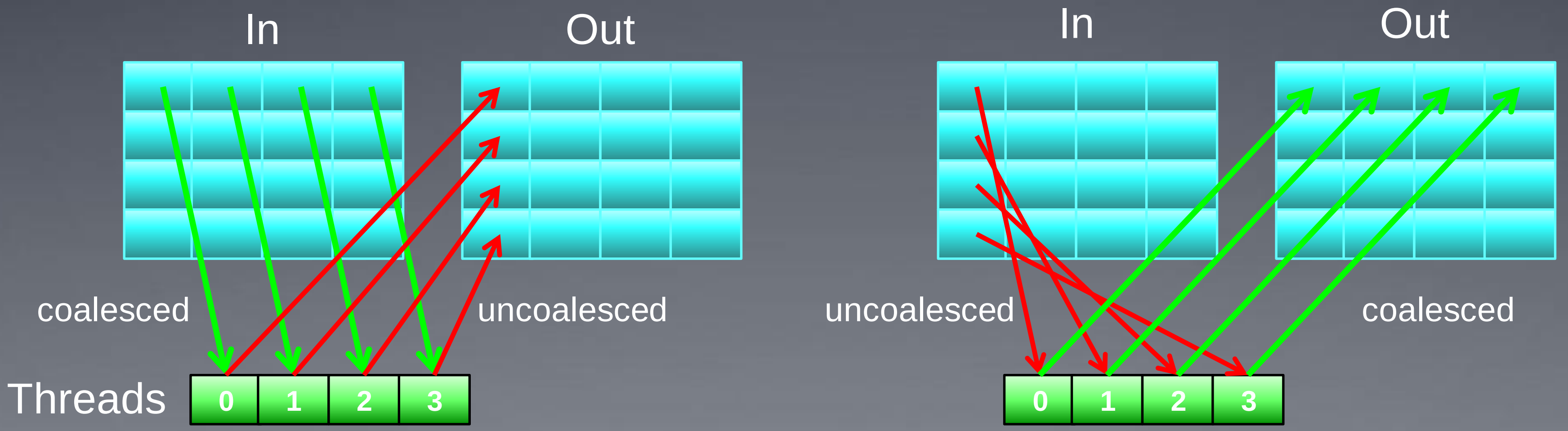
```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

Thread Mapping

- In general, threads can be created and mapped to any data element by manipulating the values returned by the thread identifier functions
- The following matrix transpose example will show how thread IDs can be modified to achieve efficient memory accesses

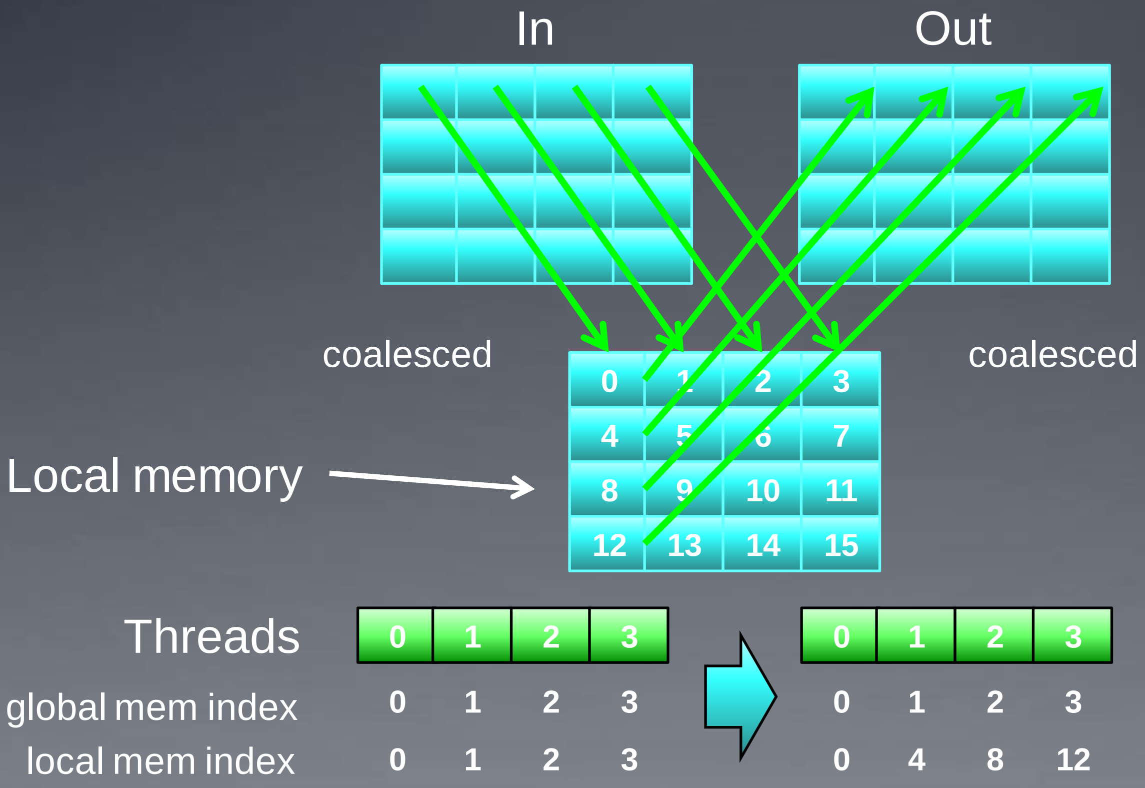
Matrix Transpose

- A matrix transpose is a straightforward technique
  - Out(x,y) = In(y,x)
- No matter which thread mapping is chosen, one operation (read/write) will produce coalesced accesses while the other (write/read) produces uncoalesced accesses
  - Note that data must be read to a temporary location (such as a register) before being written to a new location



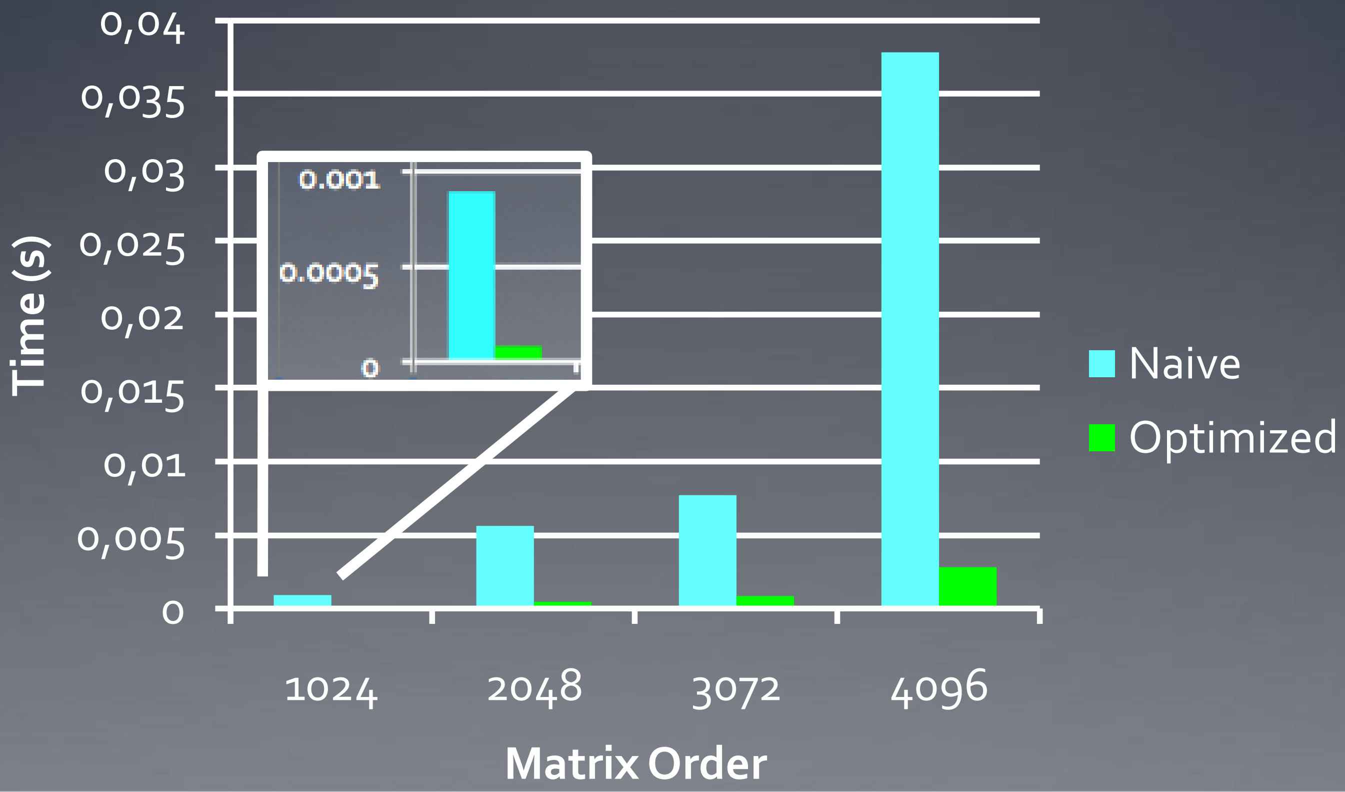
Matrix Transpose

- If local memory is used to buffer the data between reading and writing, we can rearrange the thread mapping to provide coalesced accesses in both directions
  - Note that the work group must be square



Matrix Transpose

- The following figure shows a performance comparison of the two transpose kernels for matrices of size NxM on an AMD 5870 GPU
  - “Optimized” uses local memory and thread remapping



Occupancy

- On current GPUs, work groups get mapped to compute units
  - When a work group is mapped to a compute unit, it cannot be swapped off until all of its threads complete their execution
- If there are enough resources available, multiple work groups can be mapped to the same compute unit at the same time
  - Wavefronts from another work group can be swapped in to hide latency
- Resources are fixed per compute unit (number of registers, local memory size, maximum number of threads)
  - Any one of these resource constraints may limit the number of work groups on a compute unit
- The term *occupancy* is used to describe how well the resources of the compute unit are being utilized

Occupancy – Registers

- The availability of registers is one of the major limiting factor for larger kernels
- The maximum number of registers required by a kernel must be available for all threads of a workgroup
  - Example: Consider a GPU with 16384 registers per compute unit running a kernel that requires 35 registers per thread
    - Each compute unit can execute at most 468 threads
    - This affects the choice of workgroup size
      - A workgroup of 512 is not possible
      - Only 1 workgroup of 256 threads is allowed at a time, even though 212 more threads could be running
      - 3 workgroups of 128 threads are allowed, providing 384 threads to be scheduled, etc.

Occupancy – Registers

- Consider another example:
  - A GPU has 16384 registers per compute unit
  - The work group size of a kernel is fixed at 256 threads
  - The kernel currently requires 17 registers per thread
- Given the information, each work group requires 4352 registers
  - This allows for 3 active work groups if registers are the only limiting factor
- If the code can be restructured to only use 16 registers, then 4 active work groups would be possible

Occupancy – Local Memory

- GPUs have a limited amount of local memory on each compute unit
  - 32KB of local memory on AMD GPUs
  - 32-48KB of local memory on NVIDIA GPUs
- Local memory limits the number of active work groups per compute unit
- Depending on the kernel, the data per workgroup may be fixed regardless of number of threads (e.g., histograms), or may vary based on the number of threads (e.g., matrix multiplication, convolution)

Occupancy – Threads

- GPUs have hardware limitations on the maximum number of threads per work group
  - 256 threads per WG on AMD GPUs
  - 512 threads per WG on NVIDIA GPUs
- NVIDIA GPUs have per-compute-unit limits on the number of active threads and work groups (depending on the GPU model)
  - 768 or 1024 threads per compute unit
  - 8 or 16 warps per compute unit
- AMD GPUs have GPU-wide limits on the number of wavefronts
  - 496 wavefronts on the 5870 GPU (~25 wavefronts or ~1600 threads per compute unit)

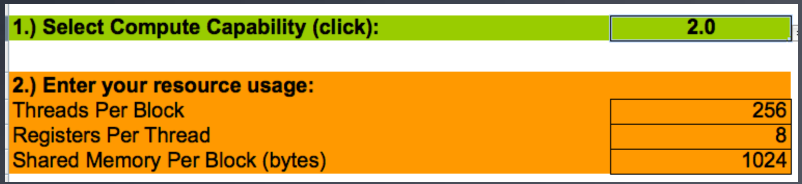
Occupancy – Limiting Factors

- The minimum of these three factors is what limits the active number of threads (or occupancy) of a compute unit
- The interactions between the factors are complex
    - The limiting factor may have either thread or wavefront granularity
    - Changing work group size may affect register or shared memory usage
    - Reducing any factor (such as register usage) slightly may have allow another work group to be active
- The CUDA occupancy calculator from NVIDIA plots these factors visually allowing the tradeoffs to be visualized

CUDA Occupancy Calculator

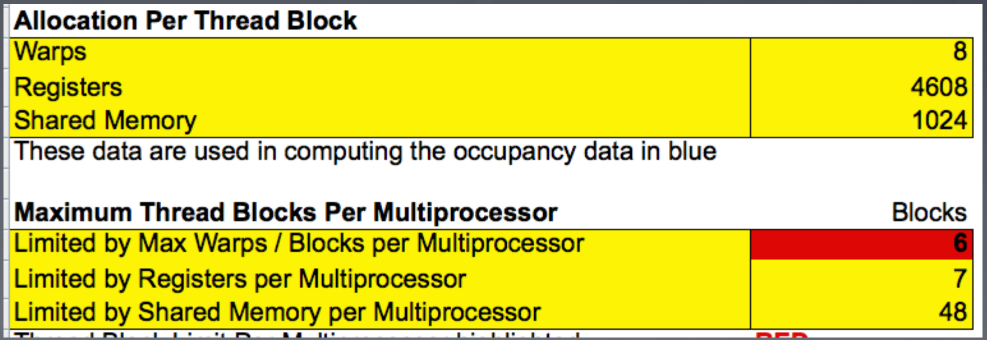
- CUDA occupancy calculator:

1. Enter hardware model and kernel requirements



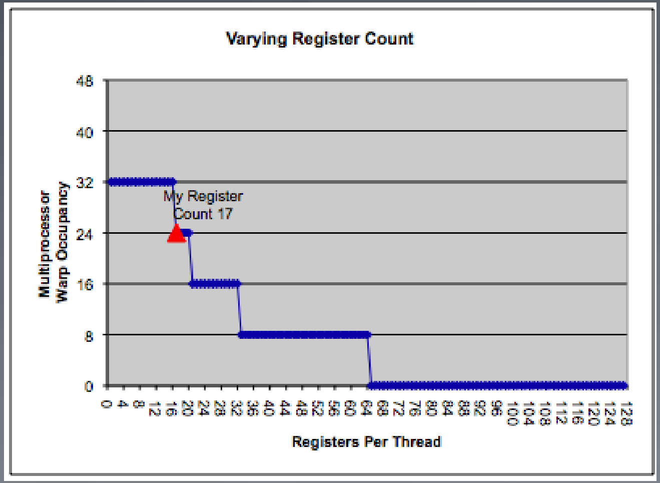
1.) Select Compute Capability (click):	**2.0**
2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	8
Shared Memory Per Block (bytes)	1024

2. Resource usage and limiting factors are displayed



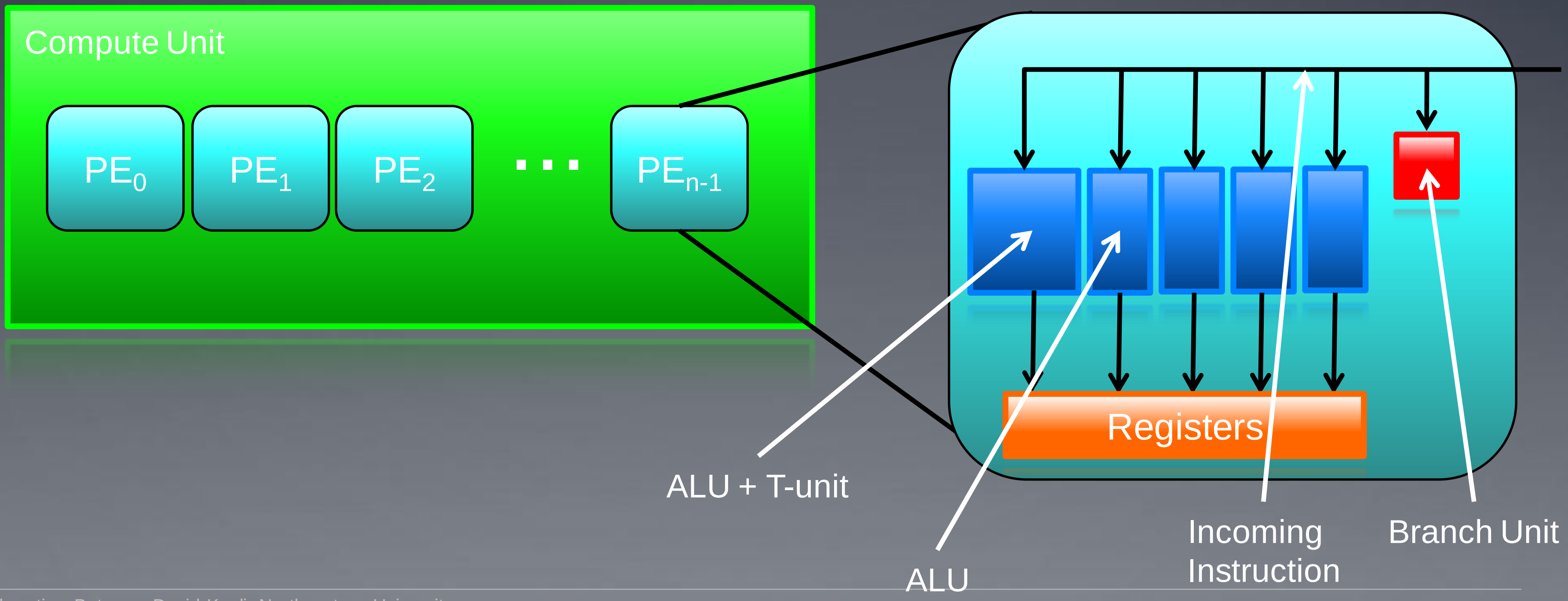
Allocation Per Thread Block	
Warps	8
Registers	4608
Shared Memory	1024
These data are used in computing the occupancy data in blue	
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	6
Limited by Registers per Multiprocessor	7
Limited by Shared Memory per Multiprocessor	48

3. Graphs are shown to visualize limiting factors



Vectorization

- On AMD GPUs, each processing element executes a 5-way VLIW instruction
  - 5 scalar operations or
  - 4 scalar operations + 1 transcendental operation

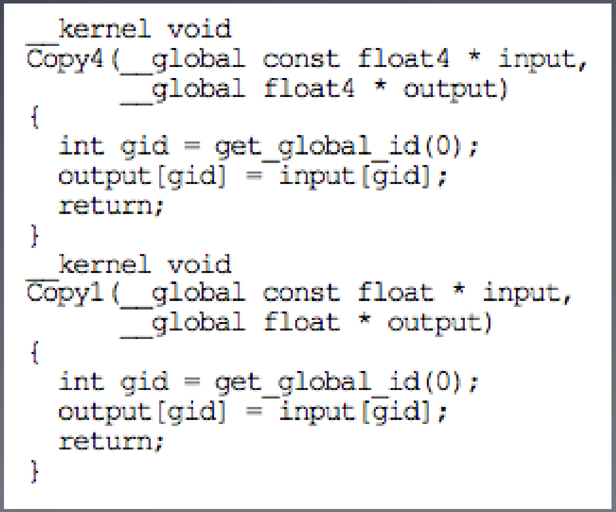


Vectorization

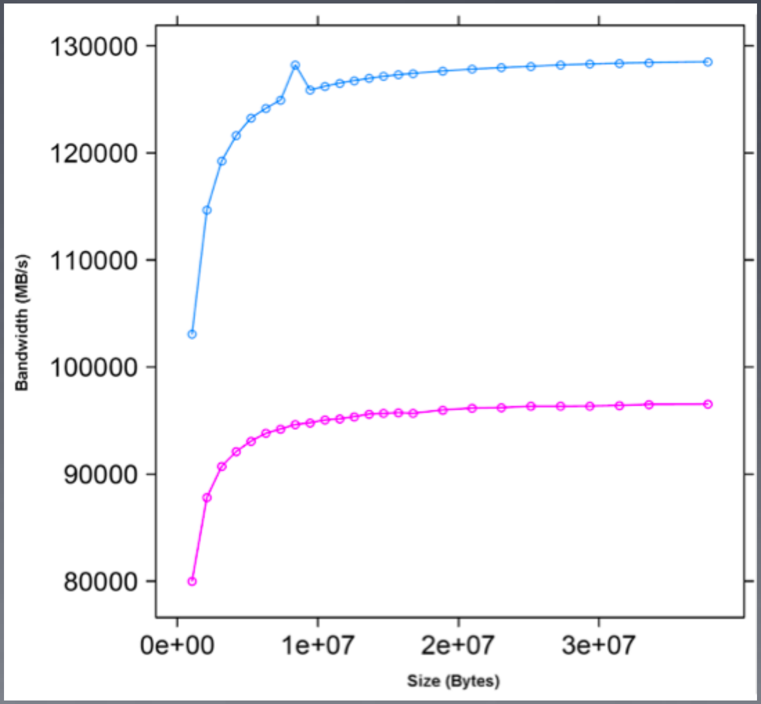
- Vectorization allows a single thread to perform multiple operations at once
- Explicit vectorization is achieved by using vector datatypes (such as `float4`) in the source program
  - When a number is appended to a datatype, the datatype becomes an array of that length
  - Operations can be performed on vector datatypes just like regular datatypes
    - Each ALU will operate on different element of the float4 data

Vectorization

- Vectorization improves memory performance on AMD GPUs
  - The *AMD Accelerated Parallel Processing OpenCL Programming Guide* compares `float` to `float4` memory bandwidth



```
__kernel void
Copy4(__global const float4 * input,
      __global float4 * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
__kernel void
Copy1(__global const float * input,
      __global float * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
```



Summary

- Although writing a simple OpenCL program is relatively easy, optimizing code can be very difficult
  - Improperly mapping loop iterations to OpenCL threads can significantly degrade performance
- When creating work groups, hardware limitations (number of registers, size of local memory, etc.) need to be considered
  - Work groups must be sized appropriately to maximize the number of active threads and properly hide latencies
- Vectorization is an important optimization for AMD GPU hardware
  - Though not covered here, vectorization may also help performance when targeting CPUs