CUDA Course

Victor Eijkhout

2024 COE 379L / CSE 392

Justification

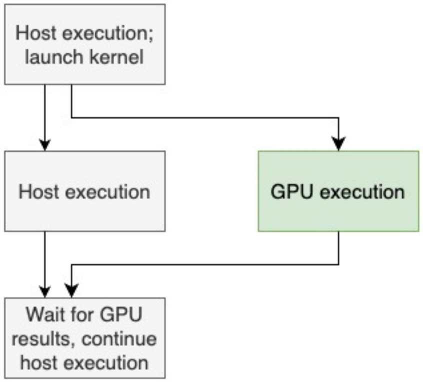
CUDA is the basic programming model for NVidia GPUs . . .
. . . which TACC is getting a lot of.

Introduction

Why GPUs

- Massive parallelism (but that's really multicore and SIMD)
- Simpler processor: less power / more computing for the same power.
- Simpler processor: no general programs, only 'data parallel' operations
  . . . such as machine learning
- Latest GPUs have very high speed on low precision
  . . . such as for machine learning

Attached processor


Host execution; launch kernel
Host execution
GPU execution
Wait for GPU results, continue host execution

- GPU is attached to regular CPU
- computation is ‘offloaded’
- GPU has its own memory:
  you may need to synchronize

Comparison

- Dozens of cores
- Core has vector instructions
- a vector instruction has ‘lanes’
- Latency hiding by caches, prefetching

- Dozens of ‘Streaming Multiprocessors’
- SM has ‘warp’s
- a warp has threads
- Latency hiding by massive multi-threading

Data parallel programming

- Kernel: operation for a single thread;
  in effect: body of inner loop
- Grid: multi-dimensional structure of available ‘threads’
  executed on cores
  (not a CPU core, more SIMD vector lane)

Loops vs CUDA

```
float *x;
for ( blockIdx_x < gridsize )
  for ( threadIdx_x < blocksize )
    size_t linear = threadIdx_x
      + blockIdx_x * blocksize;
    f( x[linear] );
```

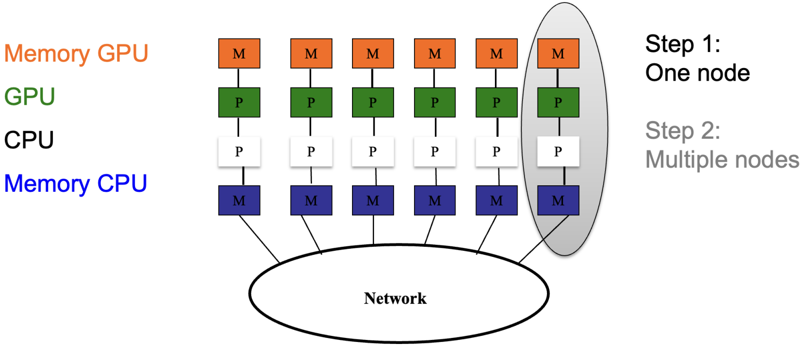
```
__global__ cu_f( float x) {
  size_t linear = threadIdx_x
      + blockIdx_x * blocksize;
  f( x[linear] );
}

float *x; // on device
cu_f<<<gridsize,blocksize>>>(x);
```

CUDA and others

- CUDA is for C and Fortran;
  limited to NVidia
- OpenACC, OpenCL: portable but tough to program
- AMD GPUs: ‘HIP’
- Intel GPUs (hm): Sycl
- OpenMP offload: universal but maybe not highest performance
- Kokkos: portable to OpenMP and all GPUs

GPU cluster

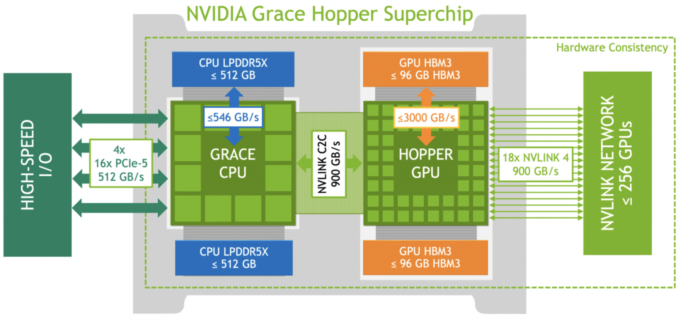


- Use GPU on a node
  (possible combined with OpenMP or other threading)
- Use MPI between nodes;
  NVlink also exists but not on large scale

TACC's Vista

- 256 Grace-Grace two-socket CPU nodes
  4 racks of 64 nodes each
- Grace-grace node has twice 72 cores: 144 total per node.
- 600 Grace-Hopper nodes
  19 racks of 32 nodes
- GH200 Grace-Hopper superchip combines
  each containing one 72-core Grace CPU (120 GiB LPDDR5X)
  and H200 GPU (96 GiB HBM3)
- GH node: 34 Tflops of FP64
  1979 Tflops of FP16 for Machine Learning.

Grace Hopper superchip



CPU vs GPU compared

Hardware Comparison

(Frontera and Lonestar)

	Dual socket Intel Cascade Lake	Dual socket AMD Milan	Nvidia H100	Nvidia A100	Nvidia A30
Core/SM	56	128	144	108	56?
Speed (GHz)	2.7	2.45	1.6	1.41	1.24
SIMD / SIMT width (32 bit)	16	8	32	32	32
Vector lanes/ CUDA cores	896*2	1024*2	16896	6912	3584
	FMA	FMA			

Kernels

Translation code to kernel

```
for ( int i=0; i<n; ++i ) {
  int threadIdx.x = i;
  f(i);
}

__global__ computef( /* fargs */ ) {
  int i = threadIdx.x;
  f(i);
}

int gridsize,blocksize;
computef<<<gridsize,blocksize>>>( fargs );
```

- Kernel is the body of the inner loop;
- Kernel is executed for each thread in the grid;
- Kernel consults `treadIdx` to find the ‘iteration’ number.

About threads

- CUDA creates thread blocks until all data has been processed
- You can tune the block size up to some limit;
- Make sure to create enough blocks.

Code:

```
// blocksize.cu
printf("Device Name: %s\n", prop.name);
printf("Max threads per block: %d\n",
       prop.maxThreadsPerBlock);
printf("Max block dimensions: %d x %d x %
    d\n",
       prop.maxThreadsDim[0],
       prop.maxThreadsDim[1],
       prop.maxThreadsDim[2]);
```

Output:

```
Device Name: NVIDIA
      ↪GH200 120GB
Max threads per block:
      ↪1024
Max block dimensions:
      ↪1024 x 1024 x 64
```

Thread indexing

```
int global_thread_id = blockIdx.x * blockDim.x + threadIdx.x; // or 2d, 3d variant
if ( global_thread_id >= datasize ) return;
```

- Global thread index from block and thread coordinate;
- Global index often used as index in the data;
- You can have more threads than data;
  no problem but you do need to test.

Data on the device

```
float
  *x, *y, *z;
size_t nbytes = N*sizeof(float);
CU_ASSERT( cudaMallocManaged( &x,nbytes ) );
CU_ASSERT( cudaMallocManaged( &y,nbytes ) );
CU_ASSERT( cudaMallocManaged( &z,nbytes ) );
```

```
// Run kernel on 1M elements on the CPU
int blocksize = 1024;
int gridsize = N/blocksize+1;
add1<<<gridsize,blocksize>>>(N,x,y,z);
// Wait for GPU to finish before accessing data on host
CU_ASSERT( cudaDeviceSynchronize() );
```

- Managed allocation is accessible on host and device
- Device is asynchronous, so synchronization needed

Utility macro

```
#define CU_ASSERT( code ) { \
  cudaError_t err = code ; \
  if (err!=cudaSuccess) { \
    printf("error <<%s>> on line %d\n",cudaGetErrorString(err),__LINE__); \
    return 1; } }
```

- CUDA routines can return error code
- Check on fatal errors.

Exercise 1

Write a CUDA kernel that adds two arrays into a third.

Using the above snippets, finish the code, test for correctness.

Two-dimensional grid

MISSING SNIPPET cublockgrid2d in codesnippetsdir=../../snippets

- Two-dimensional data calls for two-dimensional thread organization (same for three of course)
- Specify grid/block size with `dim3` objects.

Two-dimensional indexing

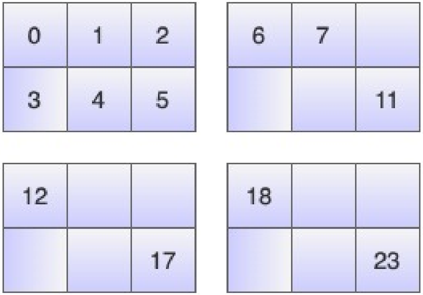
```
// index2d.cu
// Calculate the row and column indices for this thread
size_t row = blockIdx.y * blockDim.y + threadIdx.y;
size_t col = blockIdx.x * blockDim.x + threadIdx.x;

// Make sure we don't go out of bounds
if (row < height && col < width) {
// Calculate the linear index for the matrices
  size_t idx = row * width + col;

// Perform the addition
  d_c[idx] = d_a[idx] + d_b[idx];
```

- Compute row/col from `x/y` components
- Adjacent threads need to access adjacent data

Thread numbering



- Lexicographic ordering of blocks;
- Lexicographic ordering of threads in blocks.

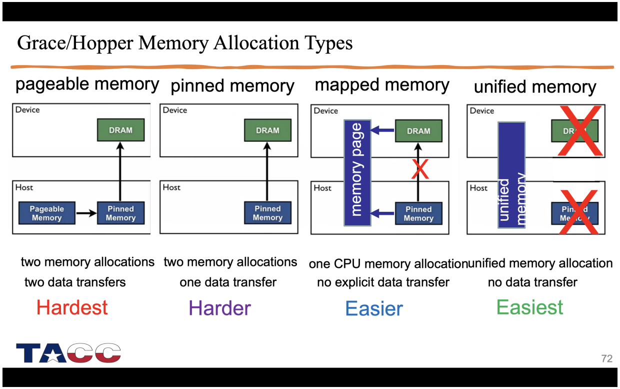
Exercise 2

Create a two-dimensional array and let each thread write its global number in the elemenent where it is active:

```
output[output_loc] = global_thread_num;
```

Memory

Host and device memory


Grace/Hopper Memory Allocation Types

pageable memory	pinned memory	mapped memory	unified memory
two memory allocations two data transfers	two memory allocations one data transfer	one CPU memory allocation no explicit data transfer	unified memory allocation no data transfer
Hardest	Harder	Easier	Easiest

- Host memory can be copied;
- can be visible on the device;
- can be unified with the device.

Explicit transfer

Allocate input data and copy host values:

```
// input allocated on device
int *dev_input;
cudaMalloc( &dev_input,bytesize );
// host values copied
cudaMemcpy( dev_input,input,bytesize,cudaMemcpyHostToDevice);
```

After the kernel execution copy output data back:

```
cudaDeviceSynchronize();
cudaMemcpy( output,dev_output,bytesize,cudaMemcpyDeviceToHost );
```

Free:

```
free(input); free(output);
cudaFree(dev_input); cudaFree(dev_output);
```

- Host holds pointer to device memory;
- used for copying in/out;
- device memory freed.

Implicit transfer

```
// add1m.cu
size_t nbytes = N*sizeof(float);
CU_ASSERT( cudaMallocManaged( &x,nbytes ) );
CU_ASSERT( cudaMallocManaged( &y,nbytes ) );
CU_ASSERT( cudaMallocManaged( &z,nbytes ) );
```

- Possibility one:
  allocate on the host with `cudaHostAlloc`
  using a flag of `cudaHostAllocMapped`.
  Pass `cudaHostGetDevicePointer` to the device.
- Easier: `cudaMallocManaged`;
  transfer is done through on-demand paging.
- (Impliciations for timing!)