CISC 372: Parallel Computing
CUDA, part 4

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

Matrix multiplication

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix} =$$

$$\begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} & a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} & a_{00}b_{02} + a_{01}b_{12} + a_{02}b_{22} \\ a_{10}b_{00} + a_{11}b_{10} + a_{12}b_{20} & a_{10}b_{01} + a_{11}b_{11} + a_{12}b_{21} & a_{10}b_{02} + a_{11}b_{12} + a_{12}b_{22} \end{bmatrix}$$

Matrix multiplication: sequential CPU code

See `seq/matmul.c`.

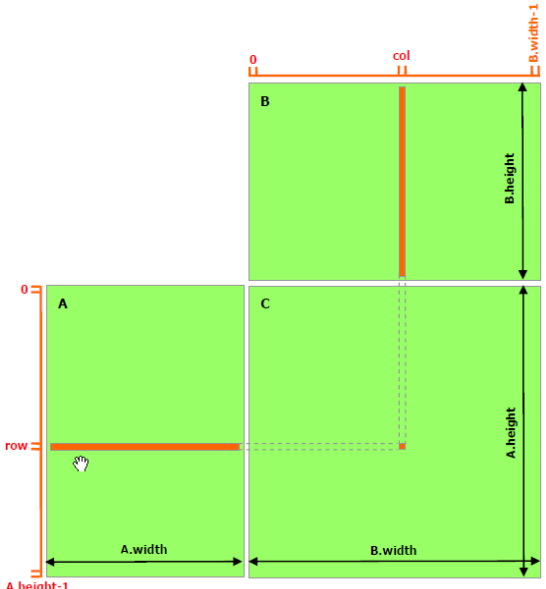
- a: $n \times l$
- b: $l \times m$
- c: $n \times m$
- $c_{ij} = \sum_{k=0}^{l-1} a_{ik} b_{kj}$

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    c[i][j] = 0.0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    for (int k = 0; k < l; k++)
      c[i][j] += a[i][k]*b[k][j];
  }
}
```

Time for $n = l = m = 8000$: > 1 hour
(512 billion multiplications)

CUDA: simple matrix multiplication [CUDA Programming Guide 3.2.3]

- A: $n \times l$
- B: $l \times m$
- C: $n \times m$
- each thread computes at most one element of c
- $c_{ij} = \sum_{k=0}^{l-1} a_{ik} b_{kj}$



Simple matrix multiplication: `matmul1.cu`: kernel

```
/* Kernel.  Multiplies a and b, sticking results into c.
   a is nxl, b is lxm, c is nxm. */
__global__ void multiply(int n, int l, int m,
                         double * a, double * b, double * c) {
  int i = blockDim.y * blockIdx.y + threadIdx.y;
  int j = blockDim.x * blockIdx.x + threadIdx.x;

  if (i < n && j < m) {
    double result = 0.0;
    for (int k = 0; k < l; k++)
      result += a[i*l + k] * b[k*m + j]; // a[i][k] * b[k][j];
    c[i*m + j] = result; // c[i][j]
  }
}
```

Simple matrix multiplication: `matmul1.cu`: kernel

```
/* Kernel.  Multiplies a and b, sticking results into c.
   a is nxl, b is lxm, c is nxm. */
__global__ void multiply(int n, int l, int m,
                         double * a, double * b, double * c) {
  int i = blockDim.y * blockIdx.y + threadIdx.y;
  int j = blockDim.x * blockIdx.x + threadIdx.x;

  if (i < n && j < m) {
    double result = 0.0;
    for (int k = 0; k < l; k++)
      result += a[i*l + k] * b[k*m + j]; // a[i][k] * b[k][j];
    c[i*m + j] = result; // c[i][j]
  }
}
```

- on Beowulf (K40c): Time for $n = l = m = 8000$: 13.8s.

Simple matrix multiplication: `matmul1.cu`: kernel

```
/* Kernel.  Multiplies a and b, sticking results into c.
   a is nxl, b is lxm, c is nxm. */
__global__ void multiply(int n, int l, int m,
                         double * a, double * b, double * c) {
  int i = blockDim.y * blockIdx.y + threadIdx.y;
  int j = blockDim.x * blockIdx.x + threadIdx.x;

  if (i < n && j < m) {
    double result = 0.0;
    for (int k = 0; k < l; k++)
      result += a[i*l + k] * b[k*m + j]; // a[i][k] * b[k][j];
    c[i*m + j] = result; // c[i][j]
  }
}
```

- on Beowulf (K40c): Time for $n = l = m = 8000$: 13.8s.
- can we do better?

`matmul1.cu`: memory analysis

Questions?

- how many times is each element of `a` read from global memory?
- how many times is each element of `b` read from global memory?

`matmul1.cu`: memory analysis

Questions?

- how many times is each element of `a` read from global memory?
- how many times is each element of `b` read from global memory?

Answers.

- element a_{ik} is used in the computation of every element of row i of c: $c_{i,0..m-1}$

`matmul1.cu`: memory analysis

Questions?

- how many times is each element of `a` read from global memory?
- how many times is each element of `b` read from global memory?

Answers.

- element a_{ik} is used in the computation of every element of row i of c: $c_{i,0..m-1}$
- hence a_{ik} is read m times

`matmul1.cu`: memory analysis

Questions?

- how many times is each element of `a` read from global memory?
- how many times is each element of `b` read from global memory?

Answers.

- element a_{ik} is used in the computation of every element of row i of c: $c_{i,0..m-1}$
- hence a_{ik} is read m times
- element b_{kj} is used in the computation of every element of column j of c: $c_{0..n-1,j}$
- hence b_{kj} is read n times

`matmul1.cu`: memory analysis

Questions?

- how many times is each element of `a` read from global memory?
- how many times is each element of `b` read from global memory?

Answers.

- element a_{ik} is used in the computation of every element of row i of c: $c_{i,0..m-1}$
- hence a_{ik} is read m times
- element b_{kj} is used in the computation of every element of column j of c: $c_{0..n-1,j}$
- hence b_{kj} is read n times
- reading from global memory is slow!

`matmul1.cu`: memory analysis

Questions?

- how many times is each element of `a` read from global memory?
- how many times is each element of `b` read from global memory?

Answers.

- element a_{ik} is used in the computation of every element of row i of c: $c_{i,0..m-1}$
- hence a_{ik} is read m times
- element b_{kj} is used in the computation of every element of column j of c: $c_{0..n-1,j}$
- hence b_{kj} is read n times
- reading from global memory is slow!
- can we use shared memory?

matmul1.cu: memory analysis

Questions?

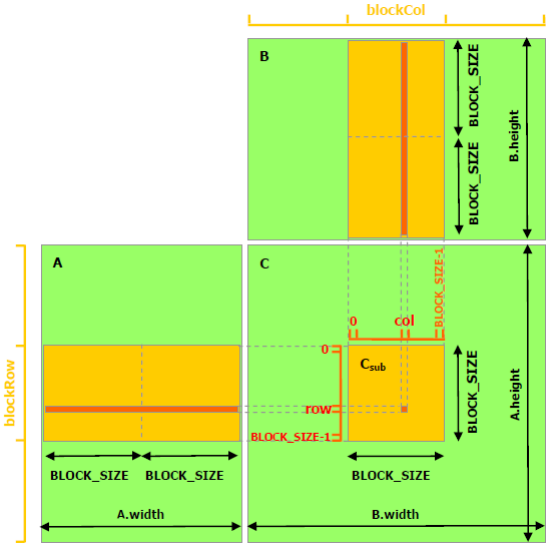
- how many times is each element of `a` read from global memory?
- how many times is each element of `b` read from global memory?

Answers.

- element a_{ik} is used in the computation of every element of row i of c: $c_{i,0..m-1}$
- hence a_{ik} is read m times
- element b_{kj} is used in the computation of every element of column j of c: $c_{0..n-1,j}$
- hence b_{kj} is read n times
- reading from global memory is slow!
- can we use shared memory?
- note: you will never be able to fit the whole matrices into shared memory at once!

Matrix multiplication: using shared memory

- C is divided into square blocks
- each CUDA block computes one block of C
  - this computation requires one row of blocks from A
  - and one column of blocks from B
- for $s = 0, 1, \ldots$
  - load block s of A's row of blocks into shared memory
  - load block s of B's column of blocks into shared memory
  - multiply to get partial result
  - accumulate sum of partial results



Using shared memory: matmul2.cu: kernel, part 1: load

```
__global__ void multiply(int n, int l, int m,
                         double * a, double * b, double * c) {
  int i_local = threadIdx.y, j_local = threadIdx.x;
  int i = blockDim.y * blockIdx.y + i_local; // row of c
  int j = blockDim.x * blockIdx.x + j_local; // col of c
  double result = 0.0;

  for (int s = 0; s < l; s += BLOCK_SIZE) {
    __shared__ double a_s[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double b_s[BLOCK_SIZE][BLOCK_SIZE];

    // each thread loads its one element of a: a[i][s+j_local]
    if (i < n && s + j_local < l)
      a_s[i_local][j_local] = a[i*l + s + j_local];
    // each thread loads its one element of b: b[s+i_local][j]
    if (s + i_local < l && j < m)
      b_s[i_local][j_local] = b[(s + i_local)*m + j];

    __syncthreads();
```

Using shared memory: matmul2.cu: kernel, part 2: multiply and store

```
    // need s + k < l, i.e., k < l - s
    const int k_stop = MIN(BLOCK_SIZE, l - s);

    for (int k = 0; k < k_stop; k++)
      if (i < n && j < m)
        result += a_s[i_local][k] * b_s[k][j_local];

    __syncthreads();
  }
  if (i < n && j < m)
    c[i*m + j] = result; // c[i][j]
}
```

Using shared memory: matmul2.cu: kernel, part 2: multiply and store

```
    // need s + k < l, i.e., k < l - s
    const int k_stop = MIN(BLOCK_SIZE, l - s);

    for (int k = 0; k < k_stop; k++)
      if (i < n && j < m)
        result += a_s[i_local][k] * b_s[k][j_local];

    __syncthreads();
  }
  if (i < n && j < m)
    c[i*m + j] = result; // c[i][j]
}
```

- on Beowulf (K40c): Time for $n = l = m = 8000$: 4.8s.

CUDA: additional concepts

- CUDA events provide a practical way to time things all withing CUDA
- multiple devices
  - `cudaGetDeviceCount(int *count)`
  - `cudaSetDevice(int device)`
  - launch a kernel on one device
  - change the device (set device to something else)
  - launch another kernel, repeat

Hybrid all the way: MPI+OpenMP+CUDA

To take advantage of all hardware in a modern supercomputer...

- use MPI for interprocess communication
- use OpenMP or other threading model for intraprocess (shared memory) concurrency on the CPU
- use CUDA to offload certain computations onto GPGPUs

...all in one program!

Hybrid all the way: MPI+OpenMP+CUDA

To take advantage of all hardware in a modern supercomputer...

- use MPI for interprocess communication
- use OpenMP or other threading model for intraprocess (shared memory) concurrency on the CPU
- use CUDA to offload certain computations onto GPGPUs

...all in one program!

- you know how to do each of these separately,
- here is simple "hello world" example as a template for combining all three forms
- `hybrid.cu`

hybrid.cu: MPI+OpenMP+CUDA, part 1: kernel

```
#include <stdio.h>
#include <mpi.h>
#include <omp.h>
#include <assert.h>

__global__ void kernel(int rank) {
  int bid = blockIdx.x;
  int tid = threadIdx.x;
  printf("Hello from block %d, thread %d of the GPU, called by process %d\n",
          bid, tid, rank);
}
```

- each MPI process will invoke the kernel
- the process will pass its rank (ID) as an argument
- this would be appropriate, for example, with one MPI process and one GPU card per node

hybrid.cu: MPI+OpenMP+CUDA, part 2: host code

```
int main (void) {
  int rank, nprocs, required = MPI_THREAD_FUNNELED, provided;
  MPI_Init_thread(&argc, &argv, required, &provided);
  assert(provided == MPI_THREAD_FUNNELED);
  MPI_Barrier(MPI_COMM_WORLD);
  double start = MPI_Wtime();
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  kernel<<<3,4>>>(rank); // 3 blocks, 4 threads per block
#pragma omp parallel shared(rank,nprocs)
  {
    int tid = omp_get_thread_num(), nthreads = omp_get_num_threads();
    printf("Greetings from CPU thread %d/%d of process %d/%d!\n",
           tid, nthreads, rank, nprocs);
  }
  cudaDeviceSynchronize();
  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0) printf("Time: %f\n", MPI_Wtime() - start);
  MPI_Finalize();
}
```

Makefile

```
NAME = hybrid
ROOT = ../../..
include $(ROOT)/common.mk
NPROCS = 2
NCORES = 4

all: $(NAME).exec

test: $(NAME).exec
	$(MPIRUN) -n $(NPROCS) -c $(NCORES) --gres=gpu:1 ./$<

$(NAME).exec: $(NAME).cu Makefile
	nvcc --compiler-options -fopenmp -o $@ --compiler-bindir mpic++ $<

.PHONY: all test
```

Run on Beowulf

```
siegel@grendel:~/372/code/src/cuda/hybrid$ make test
nvcc --compiler-options -fopenmp -o hybrid.exec --compiler-bindir mpic++ hybrid.cu
srun --unbuffered -n 2 -c 4 --gres=gpu:1 ./hybrid.exec
Greetings from CPU thread 0/4 of process 0/2!
Greetings from CPU thread 3/4 of process 0/2!
Greetings from CPU thread 2/4 of process 0/2!
Greetings from CPU thread 1/4 of process 0/2!
Hello from block 0, thread 0 of the GPU, called by process 0
Hello from block 0, thread 1 of the GPU, called by process 0
...
Hello from block 1, thread 3 of the GPU, called by process 0
Greetings from CPU thread 0/4 of process 1/2!
Greetings from CPU thread 2/4 of process 1/2!
Greetings from CPU thread 1/4 of process 1/2!
Greetings from CPU thread 3/4 of process 1/2!
Hello from block 0, thread 0 of the GPU, called by process 1
Hello from block 0, thread 1 of the GPU, called by process 1
...
Hello from block 1, thread 3 of the GPU, called by process 1
Time: 0.530748
```

Bridges SLURM script: hybrid_p100.sh

```
#!/bin/bash

# A P100 node on the GPU-small partition has: 2 GPUs, 2 16-core CPUs,
# 8 TB on-node storage.  In this configuration, 1 node is requested.
# Two MPI processes will run, each will get 16 OpenMP threads.

#SBATCH -p GPU-small
#SBATCH -t 00:01:00
#SBATCH -N 1
#SBATCH --ntasks 2
#SBATCH --gres=gpu:p100:1
# echo commands to stdout
set -x
mpirun -np $SLURM_NTASKS ./hybrid.exec
```

Bridges hybrid output

```
+ mpirun -np 2 ./hybrid.exec
Greetings from CPU thread 6/16 of process 1/2!
Greetings from CPU thread 1/16 of process 1/2!
...
Greetings from CPU thread 2/16 of process 1/2!
Greetings from CPU thread 1/16 of process 0/2!
Greetings from CPU thread 2/16 of process 0/2!
...
Greetings from CPU thread 3/16 of process 0/2!
Hello from block 0, thread 0 of the GPU, called by process 0
Hello from block 0, thread 1 of the GPU, called by process 0
...
Hello from block 2, thread 3 of the GPU, called by process 0
Time: 0.220297
Hello from block 0, thread 0 of the GPU, called by process 1
Hello from block 0, thread 1 of the GPU, called by process 1
...
Hello from block 1, thread 3 of the GPU, called by process 1
```