GPU workshop cheatsheet

OpenACC guide (PGI flags –acc –ta=nvidia –Minfo=accel)

Kernels Construct

An accelerator **kernels** construct surrounds loops to be executed on the accelerator, typically as a sequence of kernel operations.

C

#pragma acc kernels *[clause [[,] clause]…] new-line*
{ structured block }

Fortran

!$acc kernels *[clause [[,] clause]…]*
structured block
!$acc end kernels

Any data clause is allowed.

other clauses

if(*condition* **)**

When the condition is nonzero or .TRUE. the kernels region will execute on the accelerator; otherwise, it will execute on the host.

async(*expression* **)**

The kernels region executes asynchronously with the host.

Data Construct

An accelerator **data** construct defines a region of the program within which data is accessible by the accelerator.

C

#pragma acc data *[clause[[,] clause]…] new-line*
{ structured block }

Fortran

!$acc data *[clause[[,] clause]…]*
structured block
!$acc end data

Any data

Data Clauses

The description applies to the clauses used on parallel constructs, kernels constructs, data constructs, declare constructs, and update directives.

copy(*list* **)**

Allocates the data in *list* on the accelerator and copies the data from the host to the accelerator when entering the region, and copies the data from the accelerator to the host when exiting the region.

copyin(*list* **)**

Allocates the data in *list* on the accelerator and copies the data from the host to the accelerator when entering the region.

copyout(*list* **)**

Allocates the data in *list* on the accelerator and copies the data from the accelerator to the host when exiting the region.

create(*list* **)**

Allocates the data in *list* on the accelerator, but does not copy data between the host and device.

present(*list* **)**

The data in *list* must be already present on the accelerator, from some containing data region; that accelerator copy is found and used.

CUDA

Built-in kernel variables

- gridDim.[x,y,z] -> Three dimensional vector containing the dimensions of the grid. This is a constant that is set at kernel launch time. If not set explicitly each dimension defaults to 1.
- blockIdx.[x,y,z] -> Three dimensional vector containing the block index within the grid. This is a dynamic value that depends on which block calls it.
- blockDim.[x,y,z] -> Three dimensional vector containing the dimensions of the thread block. This is set at kernel launch time. If not set explicitly each dimension defaults to 1.
- threadIdx.[x,y,z] -> Three dimensional vector specifying the thread index within the thread block. Dynamic value depending on which thread calls it.

Hierarchy of Grid->Blocks->Threads

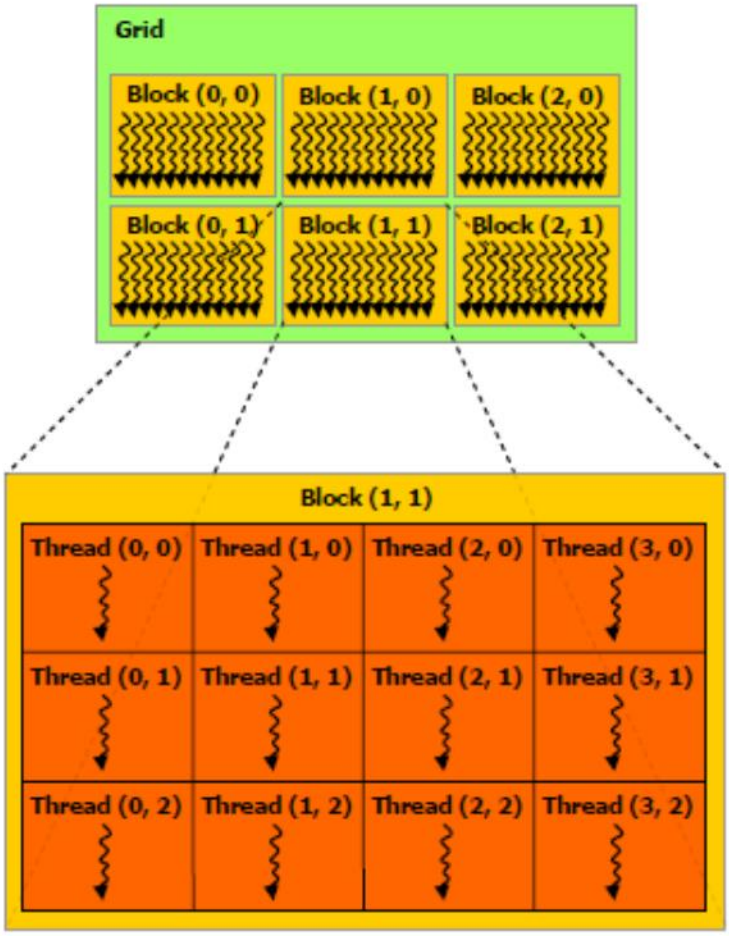

Figure 2-1. Grid of Thread Blocks

Important Functions (docs.nvidia.com)

- Kernel Launch
  - `Kernel_name<<< gridsize, blocksize >>>(arg1,arg2,…);`
- Memory Management
  - `cudaError_t cudaMalloc( void **devPtr, size_t size );`
    - Example: **`cudaMalloc( (void **) &d_c, numbytes );`**
  - `cudaError_t cudaFree( void *devPtr );`
    - Example: **`cudaFree( d_c );`**
  - `cudaError_t cudaMemcpy( void *dst, const void *src, size_t size, enum cudaMemcpyKind kind );`
    - `enum cudaMemcpyKind`
      - `cudaMemcpyHostToDevice`
      - `cudaMemcpyDeviceToHost`
      - `cudaMemcpyDeviceToDevice`
    - Example: **`cudaMemcpy( d_c, c, numbytes,cudaMemcpyHostToDevice);`**
- Error Checking
  - `cudaError_t cudaGetLastError(void);`
  - `char* cudaGetErrorString( cudaError_t code );`
  - **`printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );`**
- Streams
  - `cudaError_t cudaStreamCreate( cudaStream_t *pStream );`
  - `cudaError_t cudaStreamDestroy( cudaStream_t stream );`
  - `cudaError_t cudaMemcpyAsync( void *dst, const void *src, size_t count, cudaMemcpyKind kind, cudaStream_t stream );`
- cuBLAS
  - `cublasStatus_t cublasSetStream( cublasHandle_t handle, cudaStream_t streamId);`
  - `cublasStatus_t cublasDgemm( cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t transb, int m, int n, int k, const double *alpha, const double *A, int lda, const double *B, int ldb, const double *beta, double *C, int ldc );`
- `float atomicAdd( float *address, float val );`
  - address is the address in global memory to be updated and val is the thread local value to be added.

CUB block-wide reduction example

```
#include <cub/cub.cuh> // or equivalently <cub/block/block_reduce.cuh>
__global__ void ExampleKernel(...)
{
// Specialize BlockReduce for a 1D block of 128 threads on type int
      typedef cub::BlockReduce<int, 128> BlockReduce;

// Allocate shared memory for BlockReduce
      __shared__ typename BlockReduce::TempStorage temp_storage;

// Obtain a segment of consecutive items that are blocked across
      threads
      int thread_data[4];
      ...
// Compute the block-wide sum for thread0
      int aggregate = BlockReduce(temp_storage).Sum(thread_data);
```

CUB device-wide reduction example code

```
#include <cub/cub.cuh> // or equivalently
      <cub/device/device_radix_sort.cuh>
// Declare, allocate, and initialize device pointers for input and
      output
int num_items; // e.g., 7
int *d_in; // e.g., [8, 6, 7, 5, 3, 0, 9]
int *d_out; // e.g., [ ]
...
// Determine temporary device storage requirements
void *d_temp_storage = NULL;
size_t temp_storage_bytes = 0;

cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in,
      d_sum, num_items);

// Allocate temporary storage
cudaMalloc(&d_temp_storage, temp_storage_bytes);

// Run sum-reduction
cub::DeviceReduce::Sum(d_temp_storage, temp_storage_bytes, d_in,
      d_sum, num_items);
```

Thrust

```
thrust::host_vector<float> h_vec(size); -- allocate host vector

thrust::device_vector<float> d_vec = h_vec; allocate device
vector and perform the copy

result = thrust::reduce( d_vec.begin(), d_vec.end() ); -- call
thrust reduce on device vector
```

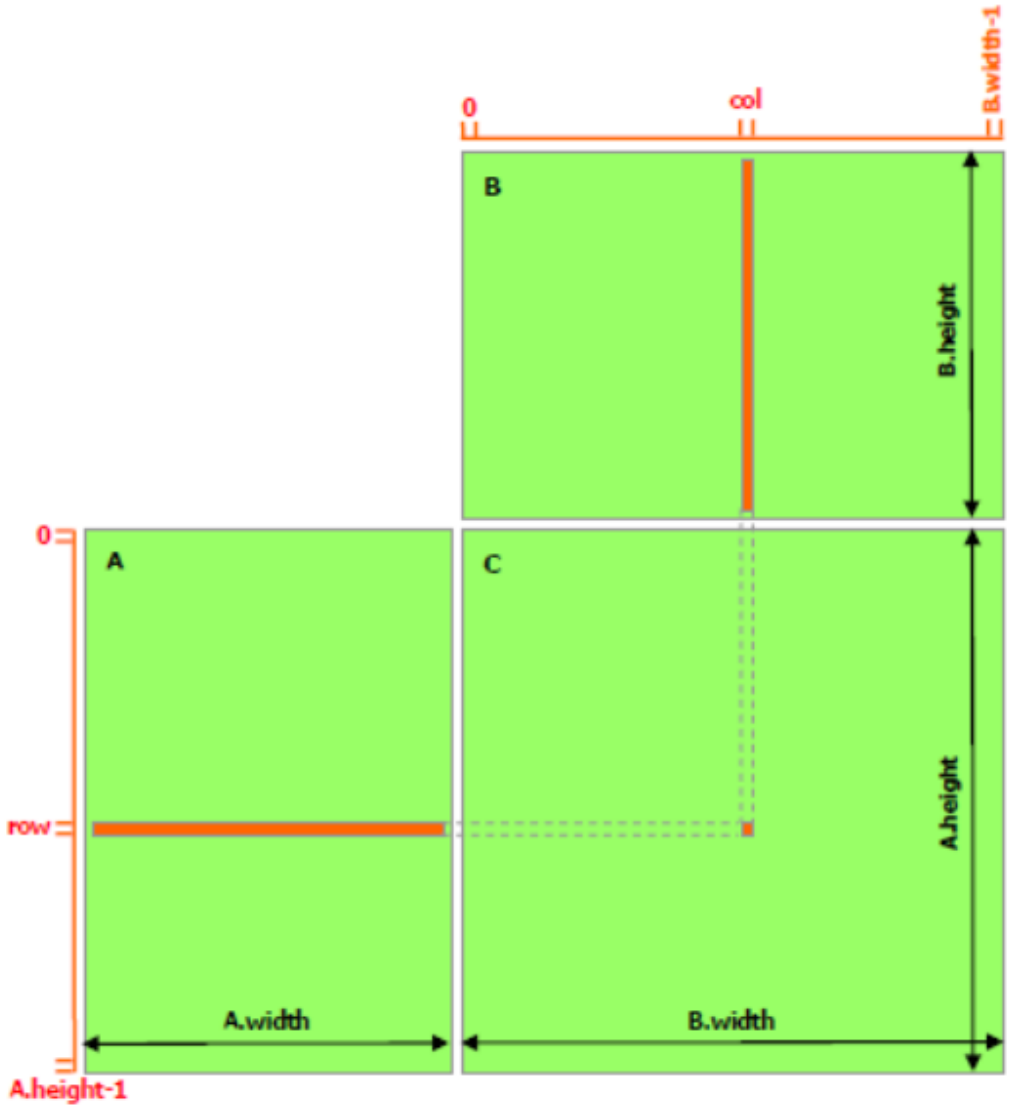

Figure 3-1. Matrix Multiplication without Shared Memory

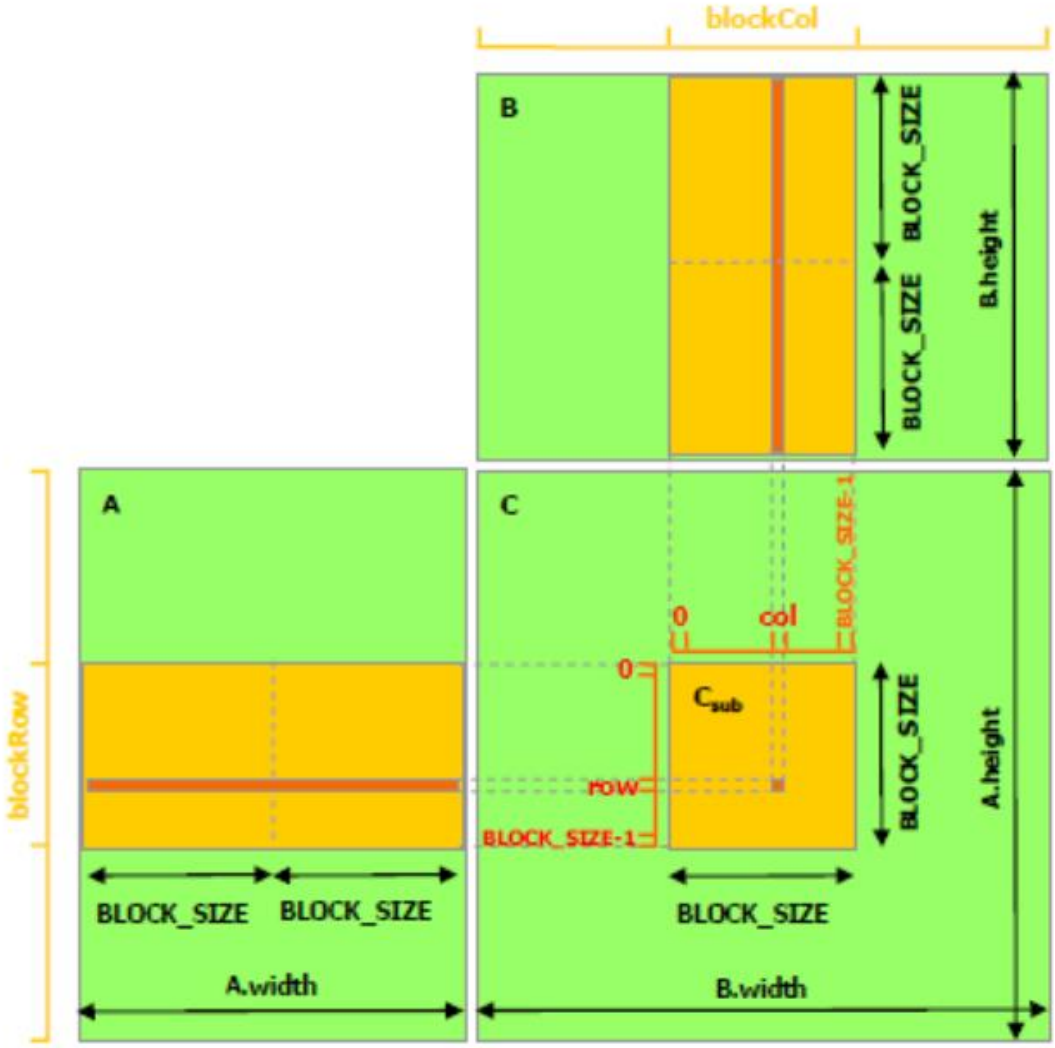

Figure 3-2. Matrix Multiplication with Shared Memory

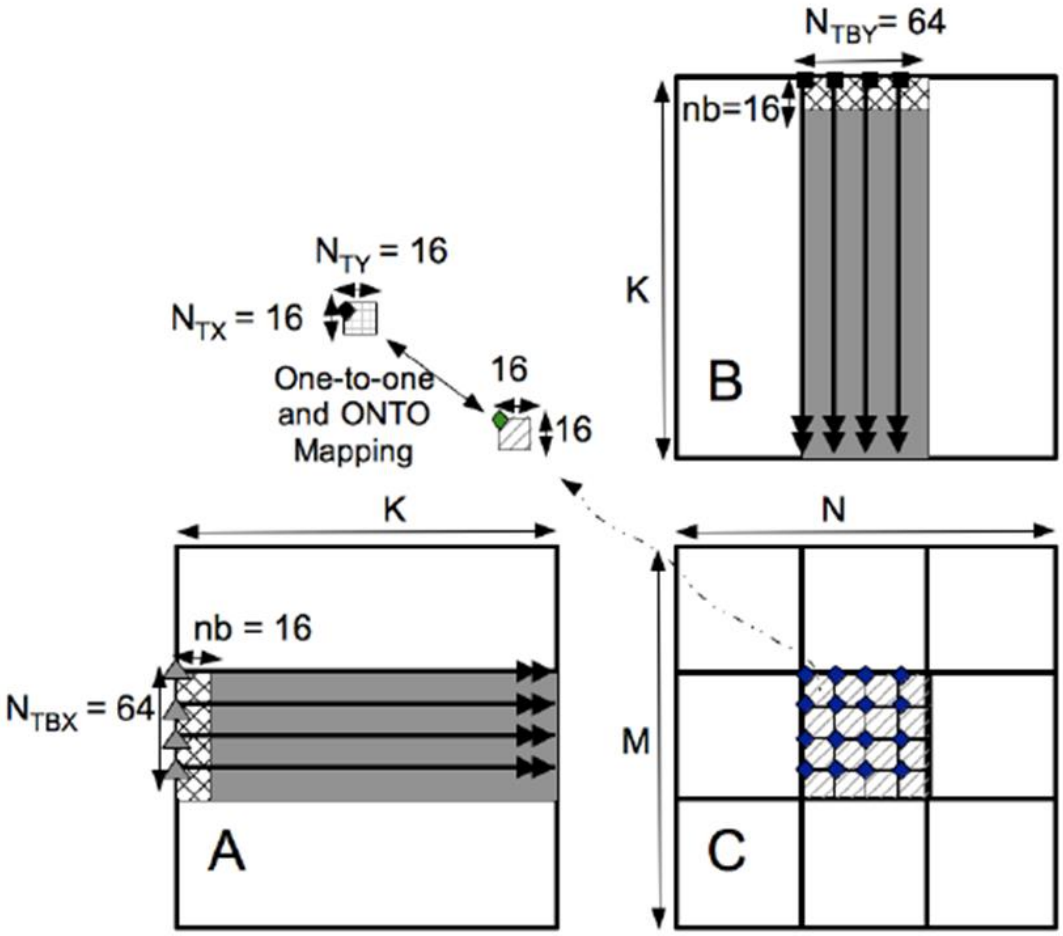

Fig. 2. The GPU GEMM ($C := \alpha AB + \beta C$) of a single TB for Fermi.

http://www.netlib.org/lapack/lawnspdf/lawn227.pdf

Instructions to connect to AWS

- Required: download/install SSH client such as Putty
    - Optional: for graphics usage. NX client such as NoMachine4. www.nomachine.com/download
        i. Use NX protocol, Password Authentication, no Proxy.
- Open a browser, go to nvlabs.qwiklab.com
    - Register (it's free) and sign in.
    - Select the correct lab and once enabled press "Start Lab".
    - Instance will take about 5 minutes to start.
- Connecting to AWS
    - SSH client
        i. Hostname and password are found on the Connection page from qwiklab
        ii. Username is "ubuntu"
    - NoMachine
        i. Create a New Connection
        ii. Use NX protocol
        iii. Password Authentication
        iv. No Proxy