GPU Programming Workshop

Session 2: Advanced topics in OpenACC and CUDA

Special Technical Projects Team

March 1, 2021

UCAR

Recap

- Introduction to GPU architecture, key concepts, and terminologies
- OpenACC programming model
  - Coding practice
- CUDA programming model
  - Coding practice
- OpenACC and CUDA coding practice (naive matrix multiplication)

Before going further, clone the workshop code, slides, and reference materials using:
git clone https://github.com/NCAR/GPU_workshop.git

Overview

- Fused multiply add (FMA) OpenACC programming model
  - Coding practice
- CUDA programming model
  - Coding practice
- OpenACC and CUDA coding practice (naive matrix multiplication)

Before going further, clone the workshop code, slides, and reference materials using:
git clone https://github.com/NCAR/GPU_workshop.git

Naive FMA with OpenACC

Previously, we examined a matrix addition and naive matrix multiplication code. FMA combines both of these operations into a single function call, the sequential version of which is shown below. As a review, we will be adding the necessary OpenACC parallelization directives to this function.

```
//FMA operation
void CPU_FMA(float *A, float *B, float *C, float *D, const int rows, const int cols)
{
        float fSum;
        for (int i=0; i<cols;i++)
        {
                for(int j = 0; j<rows; j++)
                {
                        fSum = 0.0f;
                        for(int k=0; k<rows; k++)
                        {
                                //linear adressing multiplication
                                fSum +=(A[(i*rows)+k]*B[(k*rows)+j]);
                        }
                        //Addition
                        D[(i*rows)+j] = fSum + C[(i*rows)+j];
                }
        }
}
```

Naive Multiplication

Matrix Addition

A recap of some of the important OpenACC pragmas is on the following slides

OpenACC Directives/Constructs

Construct	C code	Description
Kernels	#pragma acc kernels [clauses]	Surrounds loops to be executed on the GPU
Parallel	#pragma acc parallel [clauses]	Launches a number of gangs in parallel each with a number of workers, each with vector or SIMD operations
Loop	#pragma acc loop [clauses]	Applies to the immediately following loop or nested loops and describes the type of parallelism to execute these loops on the GPU
Data	#pragma acc data [clauses]	Defines a region of the program within which data is accessible by the GPU
Host data	#pragma acc host_data [clauses]	Makes the address of the device data available on the host
Cache	#pragma acc cache (list)	Added to the top of the loop. The elements in the list are cached in the software managed data cache
Update	#pragma acc update [clauses]	Copies data between the host memory and the data allocated on the GPU memory, and vice-versa
Clause - Collapse	#pragma acc loop collapse(n)	The collapse clause is used to specify how many tightly nested loops are associated with the loop clause.
Clause - Reduction	#pragma acc loop reduction(+:temp)	The reduction clause specifies a reduction operator and one or more vars.

OpenACC Recap

Data construct

- *#pragma acc data [clauses]* defines a region below it within which the data is accessible by the GPU.

Clause	Description
copy(list)	1. Allocates the data in list on the GPU 2. Copies the data from the host to the GPU when entering the region 3. Copies the data from the GPU to the host when exiting the region
copyin(list)	Allocates the data in list on the GPU and copies the data from the host to the GPU when entering the region.
copyout(list)	Allocates the data in list on the GPU and copies the data from the GPU to the host when exiting the region.
create(list)	Allocates the data in list on the GPU, but does not copy data between the host and device.
present(list)	The data in the list must be already present on the GPU from some containing data region, which then can be found and used.

The information in this table and more about different directives and clauses at:
https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf

Structured v. Unstructured Directives

Structured:

- Regions are defined by **{ }**
- Implicit variable lifetime marked
- Most up-to-date data present on GPU with fresh copies

Unstructured:

- Data regions are not defined with braces
- Data moved onto GPU remains until it is manually cleared
- Data which is not modified on the CPU may be reused across parallel loops without additional data movement clauses
- Additional unstructured data directives need to be added to define the lifetime of variables

Unstructured Data Directives

Unstructured Data Directives:

enter data - defines the start of an unstructured data lifetime

clauses : copyin(list), create(list)

exit data - defines the end of an unstructured data lifetime

clauses: copyout(list), delete(list)'

```
#pragma acc enter data copyin(a[0:N],b[0:N]) create(c[0:N])

  #pragma acc parallel loop
  for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
  }

#pragma acc exit data copyout(c[0:N])
```

OpenACC Present Directive

When data scope is declared outside of the compute scope, the present directive tells the compiler that the data is already on the GPU.

```
function main( )
{
  #pragma acc data copy(A)
  example_function(A, n, m)
}
```

```
example_function( double[N][M] A , n , m )
{
  #pragma acc data present(A[n][m]) create(Anew)
  while (err < tol && iter < iter_max)
  {
     ...
  }
}
```

Breakout rooms

Naive FMA with OpenACC

Navigate to the GPU_workshop/Lesson_3_FMA/OpenACC/exercise folder

Exercise 5: Naive FMA Using OpenACC

First, the matrices need to be copied over to the GPU in the **main.cpp** file.

- Add a directive that provides a copy of matrix A, B, and C for the device to manipulate

Next, In the **matrix_mult.cc** file, we will be adding OpenACC directives to the openacc_matrix_mult function.

- Add **loop directives** with collapse clause
- Give compiler hint that matrix data is already **present** on the GPU

Fused-Multiply Add Operator

Fused multiply add instructions (also known as floating-point contractions) allow the compiler to execute the operation

d = round(a*b+c)

where the value is rounded to the variable size allotted for d. This operation is generally automatically generated without extra prompting.

Example output with FMA

It is important to be aware of the difference this rounding operation may generate from the intended outcome. without FMA enabled, the operation would execute

d = round(round(a*b) + c)

Example output without FMA

Though subtle, this difference can cause values to break tolerance when run on different machine configurations and between CPU and GPU operations.

Fused-Multiply Add Operator

Disabling FMA Operations:

Within the Makefile, the following flags must be added to the build:

CPU Flag: Mnofma

GPU Flag: nofma

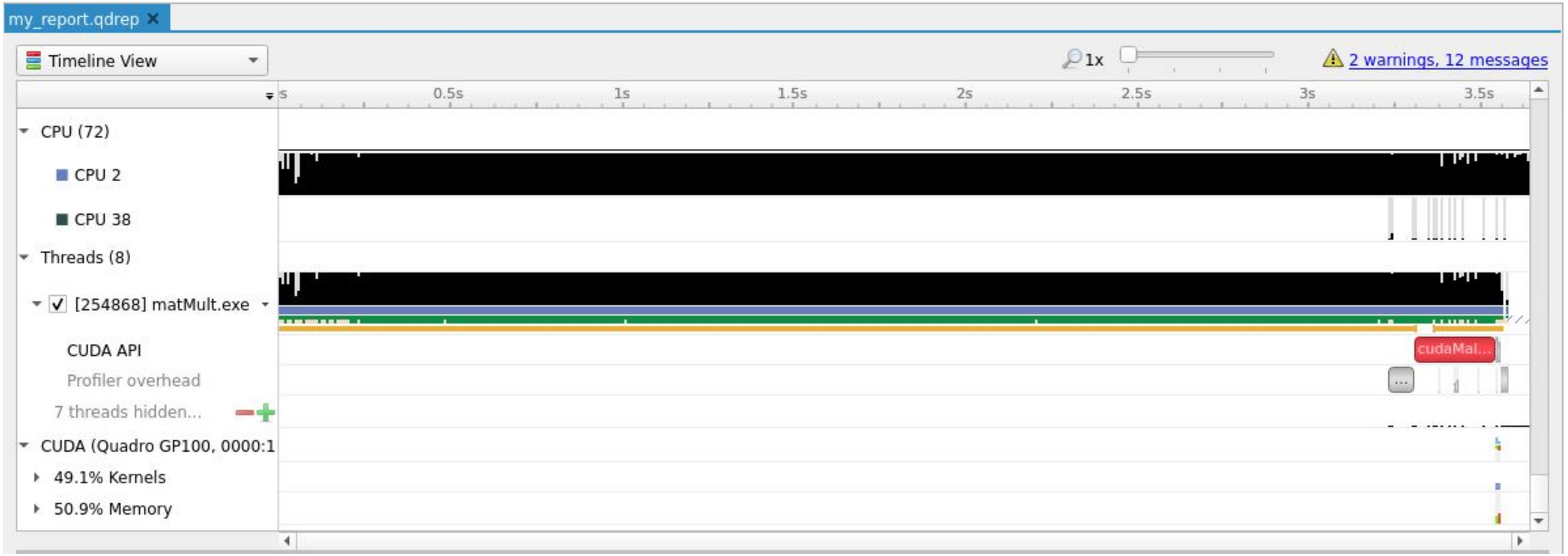
```
# Compilers and linkers
CC=nvc++

# Compilation flags
CFLAGS= -g -O3 -std=c++11 -Wall

ifeq ($(OPENACC)$(useFMA),truefalse)
ACCFLAGS= -acc -gpu=cc60,cc70,managed,nofma -Minfo=all
ADDFLAGS= -Mnofma
ifeq ($(OPENACC)$(useFMA),truetrue)
ACCFLAGS= -acc -gpu=cc60,cc70,managed -Minfo=all
ADDFLAGS=
else
ACCFLAGS=
ADDFLAGS=
endif
```

Note:
ADDFLAGS will need to be added to the Makefile binarys and output commands as well (wherever the ACCFLAGS is also seen)

Nsight Profiler

Last session, we looked at the nvprof which is a command line profiler that prints out information about your program execution.

Today, we will be looking at Nsight which is also a profiler but is more user friendly as it displays information about your program execution on an interactive GUI as seen below.



Using Nsight Profiling on Casper

Because we will be using the Nsight GUI, this exercise will require more steps than was required to use the nvprof command line profiler.

Step 1: Install TigerVnc viewer on your host computer. Information about installation can be found at: https://tigervnc.org/

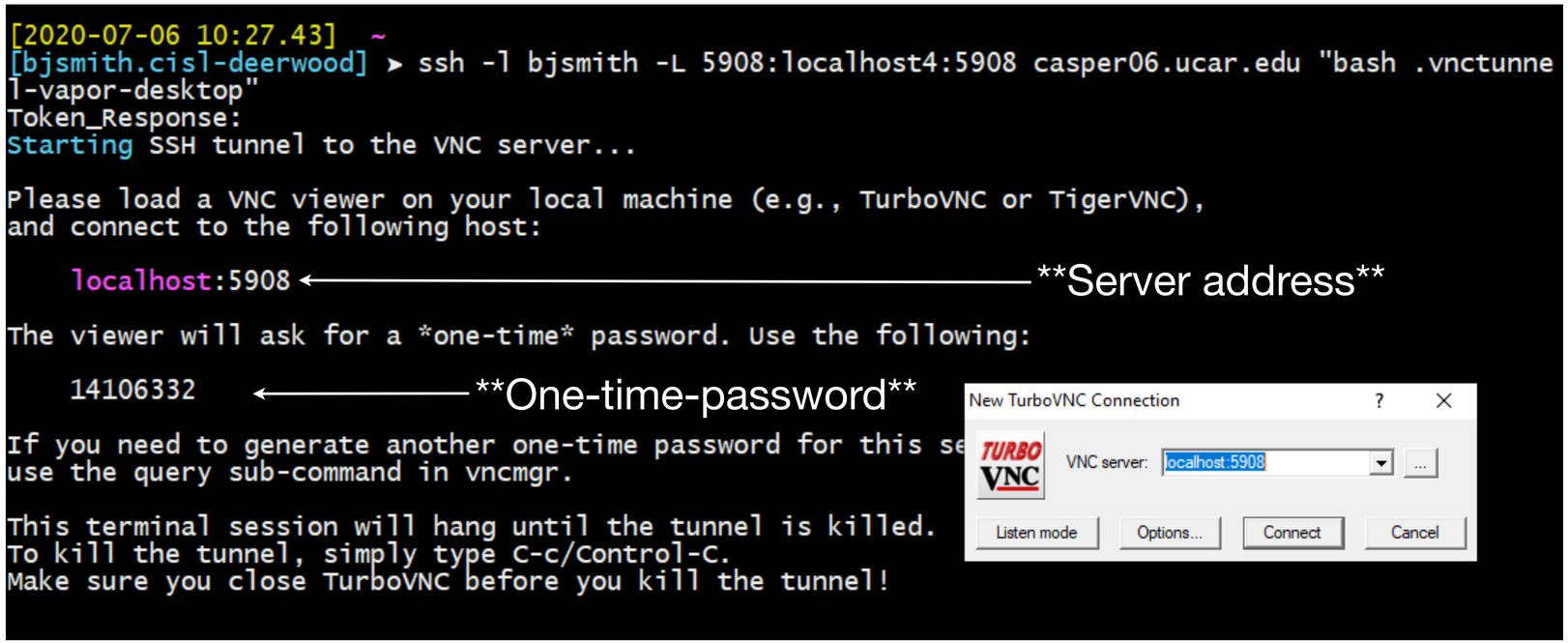
Step 2: Login to Casper and run the commands below. After running these commands, your screen should look like the image below.

- module load nvhpc.
- vncmgr list
- vncmgr create [session name] --account [Project]

```
To activate the VNC session on your client machine (external network):

  1) Create an SSH tunnel to stream desktop data to your local computer:

       ssh -l bjsmith -L 5908:localhost4:5908 casper06.ucar.edu "bash .vnctunnel-vapor-desktop"

  2) Follow instructions given in local terminal. The local terminal session
     will hang after creating the tunnel. This behavior is normal.

If you disconnect from the VNC server and wish to reconnect,
you will need to restart the tunnel to generate a new password.
This VNC server will timeout and shut down automatically after 7:59:59.

To end the Slurm job and shut down the VNC server:

  1) Use scancel to end the VNC server job:

       scancel 5530022

Note: you can avoid the tunneling step if using the NCAR internal network
      or the NCAR VPN (https://www2.cisl.ucar.edu/user-support/vpn-access)
```

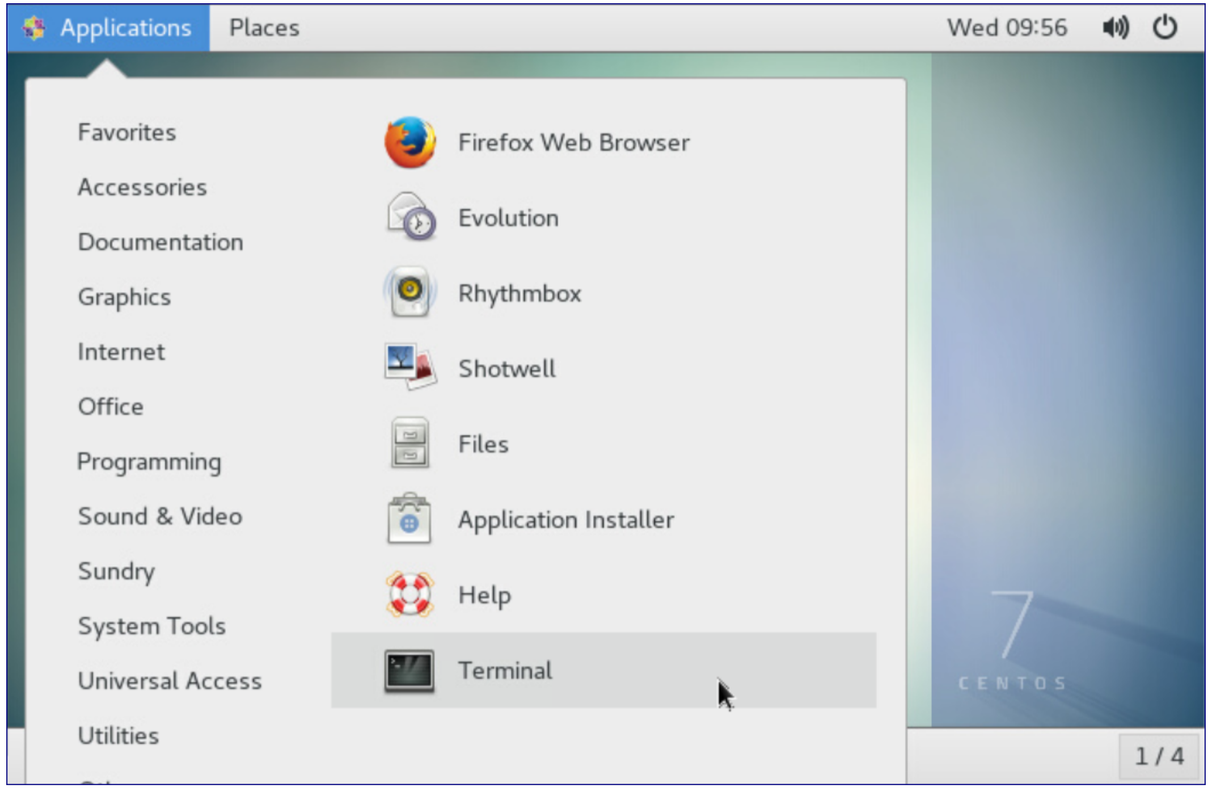
Step 3: On your host computer, run the ssh command gotten from the output of the previous step.
Note: Your host computer must be connected to the NCAR Internal network or the NCAR VPN.

Step 4: Open Tigervnc viewer and enter the Server address and the "one-time" password given to you after running the ssh command.



Using Nsight Profiler on Casper

Step 5: Once you are logged in, open the terminal window.



Using Nsight on Casper

Step 6: Once you are on the terminal, load the cuda module using "module load cuda". Without loading this, the nsys commands will not be loaded into our environment.

Step 7: Run the build script and submit the job.

Now we profile!

Step 8: Run the following command to profile your code:

"nsys profile -o profileData ./exec_file.exe"

Step 9: Finally, run the command "nsight-sys profileData.qdrep" to view your profile on the GUI

Exercise: FMA v. No FMA

1. Modify the Makefile to include the noFMA flag options for both the CPU and GPU calculations

2a. Change the build script to set useFMA=false

2b. Run the program and note the values printed out.
- It is recommended to copy these values into a scratch file to compare with the FMA enabled value

3a. Adjust the build script to useFMA=true

3b. Run the program and note the values printed out.
- Are there any differences in the calculated values?

Exercise: OpenACC C/C++ FMA Takeaways

- Structured vs Unstructured Data Constructs
- compiler flag to disable (-nofma)
- Profiler GUI

Tensor cores

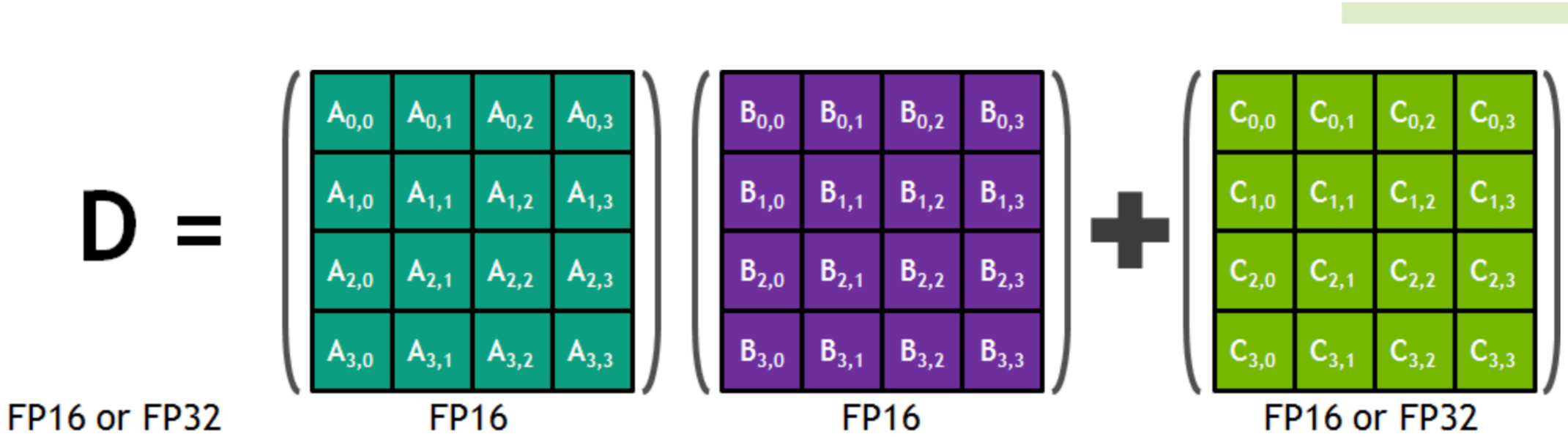


Figure 1: Tensor Core 4x4x4 matrix multiply and accumulate.

How to program tensor cores?

- CuBLAS library
- Using functions from nvcuda class

https://github.com/wzsh/wmma_tensorcore_sample/blob/master/matrix_wmma/matrix_wmma/main.cu

https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/#:~:text=Tensor%20Cores%20enable%20AI%20programmers,%2C%20MXNet%2C%20and%20Caffe2).

Tensor cores

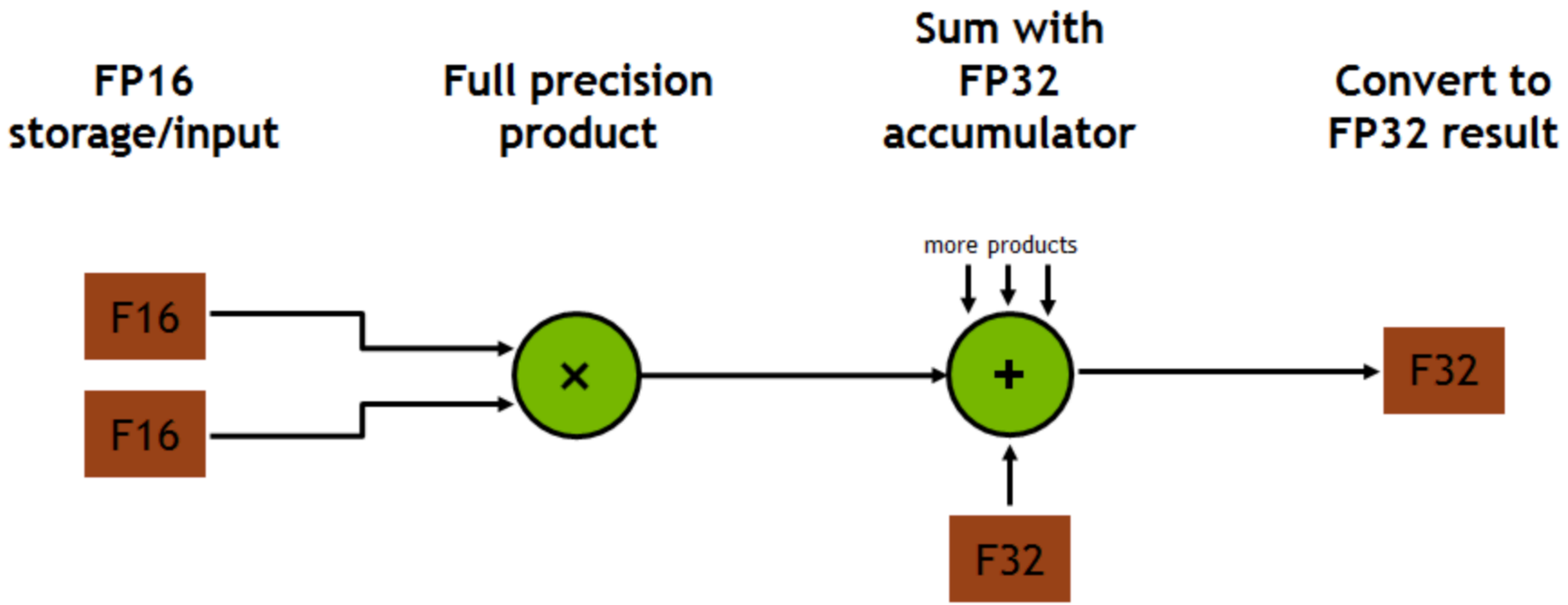
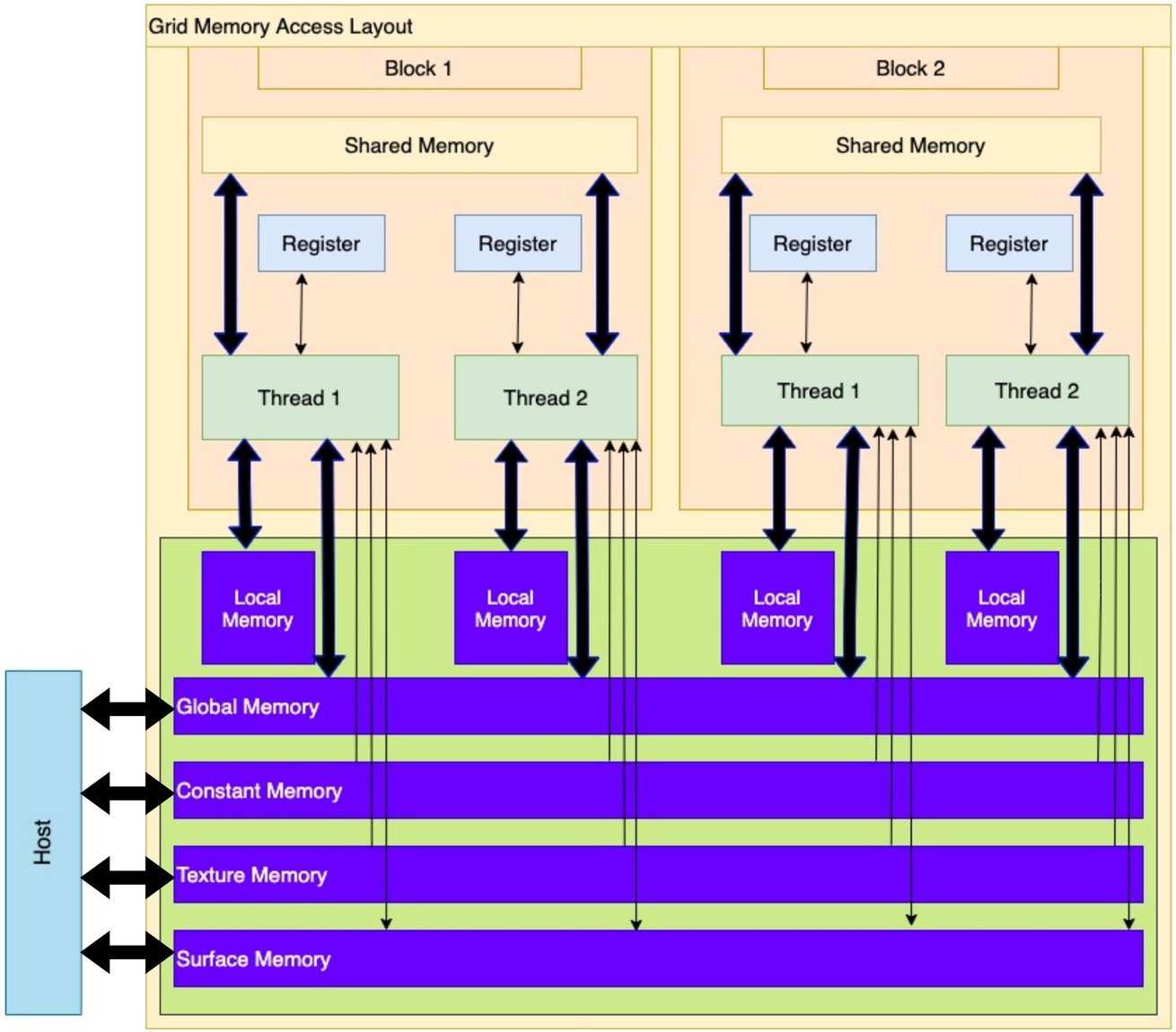


Figure 2: Volta GV100 Tensor Core operation.

Demo

GPU Memory Architecture



For K2200 GPU

Memory Type	Size	Latency (clock cycles)
Register	64 kB	1
Shared Memory	16 to 48 kB	1 to 32
Global Memory	4 GB	400 to 600

GPU Memory Architecture - Another View

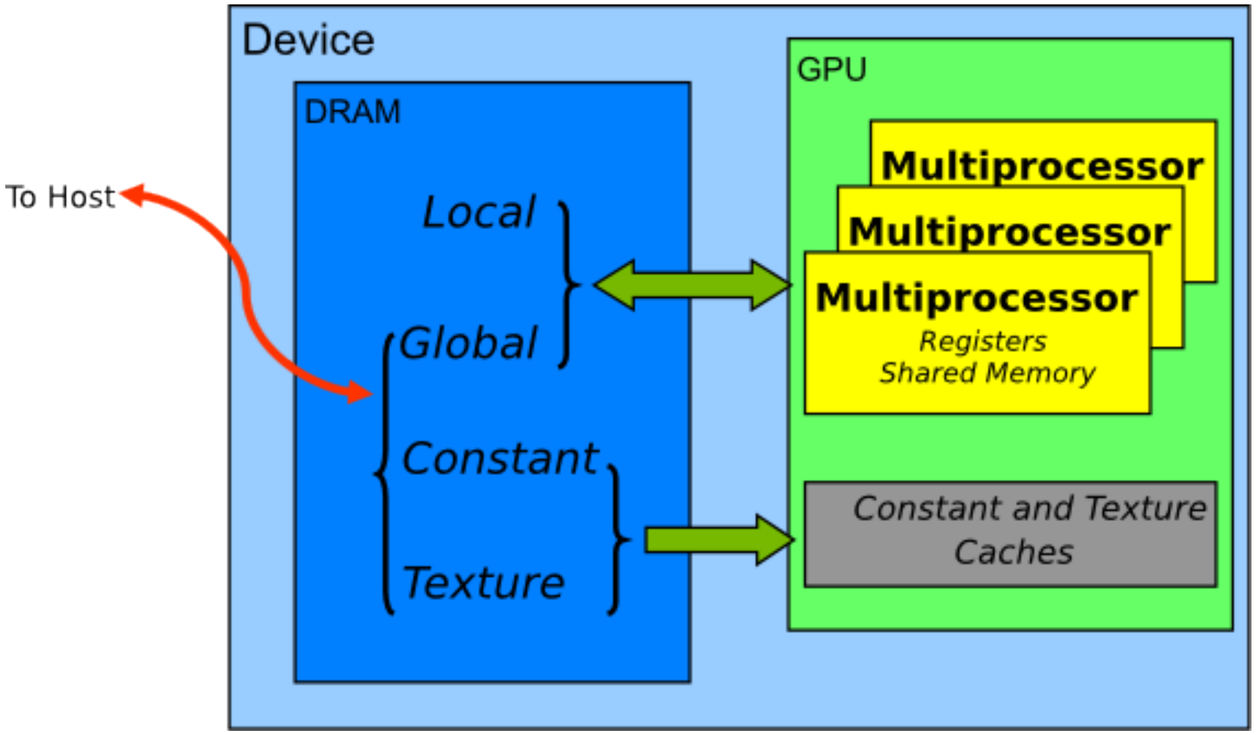


Table 1. Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

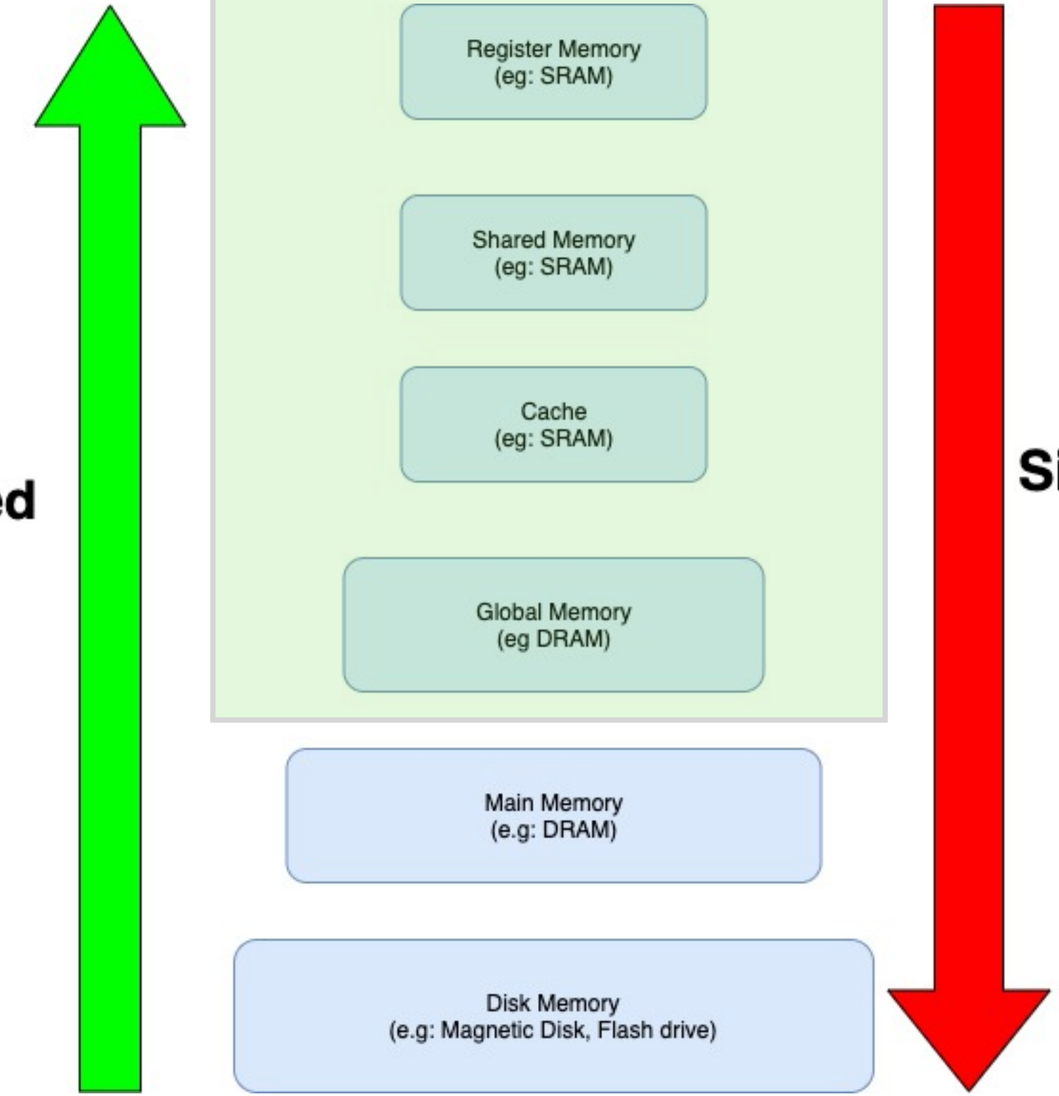
† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

Source: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

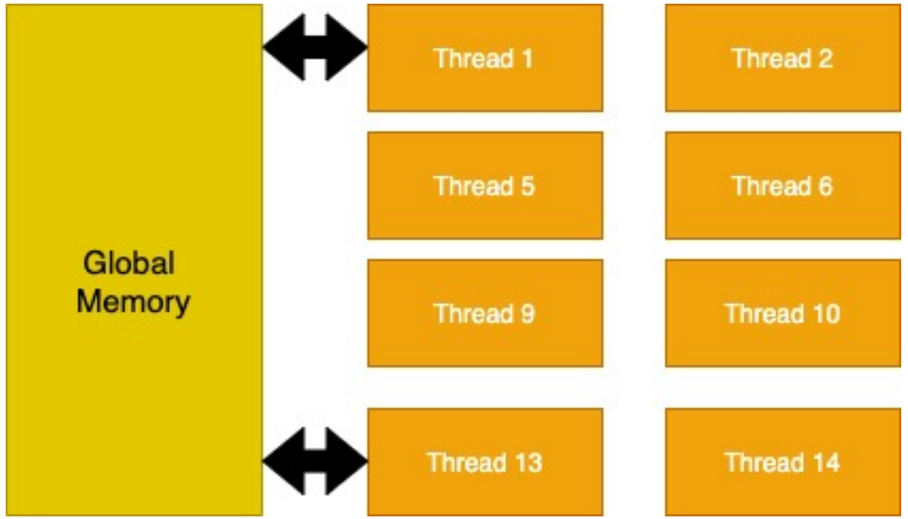
GPU Memory Hierarchy

Latency in clock cycles		Size
1	Register Memory (eg: SRAM)	64 KB
1 to 32	Shared Memory (eg: SRAM)	16 - 48 KB
Varies	Cache (eg: SRAM)	64 KB
400 to 600	Global Memory (eg DRAM)	4 GB
	Main Memory (e.g: DRAM)	
	Disk Memory (e.g: Magnetic Disk, Flash drive)	



Latency and size are for K2200 GPU

Global Memory



Size typically ranges from 4 up to 32 GB

What? Memory communication data between multiprocessors and device

How? __global__ float commondata[maxMem];

Who? All threads on the device, host, other GPUs (UVA)

Scope? Lifetime of application

Access? R+W

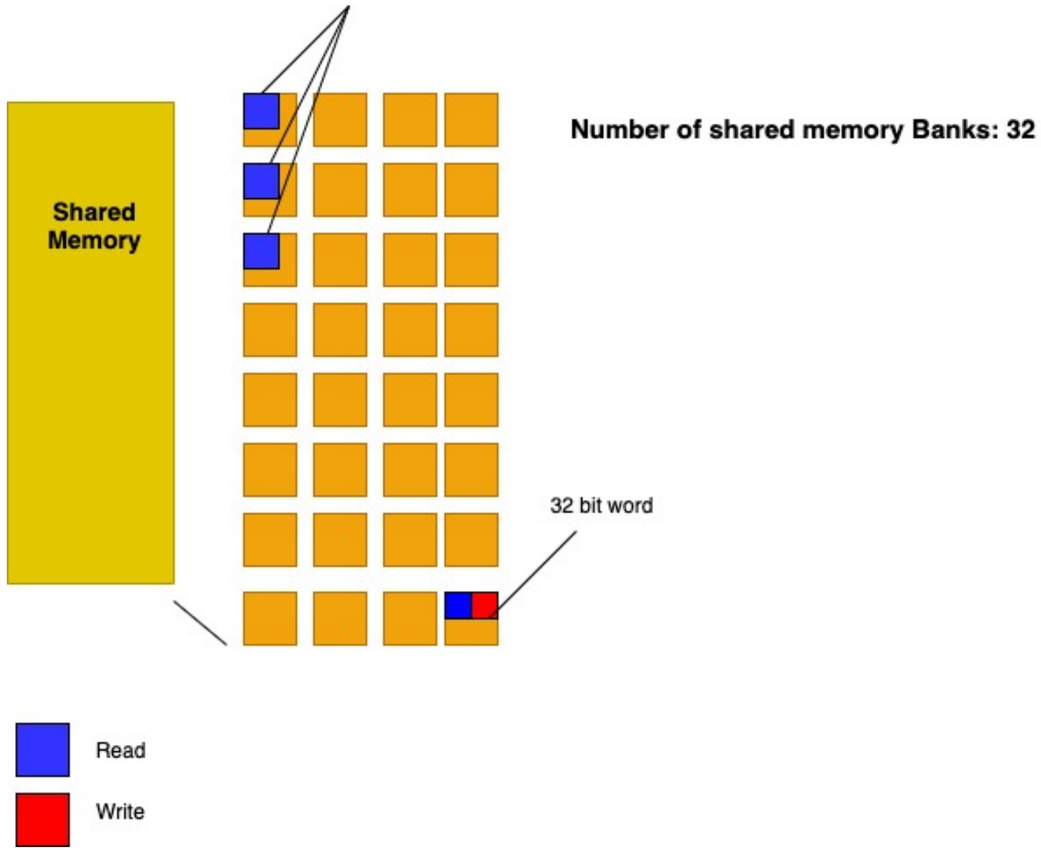
Problems? Bad alignment of data can slow down even further

Registers and Local Memory

Number of 32-bit registers per multiprocessor: 64k (except CC 3.7: 128k)

What?	Register is the memory for the ALUs, so it is on-chip and fast!
How?	Declare a variable, e.g: int counter;
Who?	A register is assigned to a single thread only
Scope?	Lifetime of thread
Access?	Read+Write, thread-private
Problems?	Affects occupancy and if a thread wants too many registers, they become spilled out to local memory (slow)

Shared Memory & L1 Cache



- Shared memory banks are equally sized memory modules, can be accessed simultaneously
- On the V100, the combined L1 cache and shared memory capacity is 128KB for each block
- Shared memory is configurable up to 96KB
- Programs that don't use shared memory can use all 128KB as L1 cache

Source: https://developer.nvidia.com/blog/inside-volta/

CUDA Shared Memory

Allocate shared memory with __**shared**__ keyword

```
// Statically allocate a tile of shared memory.
__shared__ float s_a[SHMEM_SIZE];
__shared__ float s_b[SHMEM_SIZE];
```

We want to make use of shared memory's speed for data that will be reused within a block. We also want to avoid inefficient memory access patterns from using global memory.

Data should be loaded into shared memory in a way that enables efficient memory accesses.

Breakout rooms

Navigate to the **GPU_workshop/Lesson_4_SharedMem/CUDA/exercise** folder

Exercise: Matrix Multiplication with CUDA Shared Memory

Kernel Steps

- Compute each thread's global row and column index.
- Statically allocate a tile of shared memory. Tile size should equal the number of threads per block.
- Declare a temporary variable to accumulate calculated elements for the C matrix.
- Sweep tiles of size blockDim.x across matrices A and B. For matrix A, keep the row invariant and iterate through columns. For matrix B, keep the column invariant and iterate through rows.
- Load in elements from A and B into shared memory for each tile.
- Wait for tiles to be loaded in before doing computation.
- Do matrix multiplication on the small matrix within the current tile.
- Wait for all threads to finish using current tiles before loading in new ones.
- Write resulting calculation as an element of the C matrix.

To build use **./build.sh**

To submit use **sbatch submit.sh**

Exercise: Matrix Multiplication with CUDA Shared Memory Check

Compare the execution times of the different matrix multiplication implementations we've done so far.

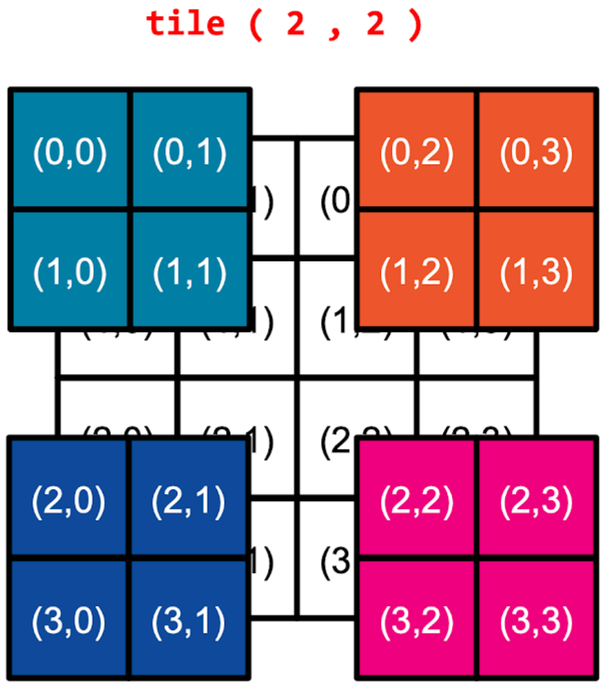
Which implementation had the fastest GPU execution time for multiplying 1024 x 1024 matrices?

When might we see better performance using shared memory?

tile Clause

- tile(x, y, z, ...)
- Breaks multidimensional tightly nested loops into “tiles”
- Can increase data locality in some codes, exploit caching
- Execute multiple “tiles” simultaneously
- Product of dimensions must be less than or equal to maximum number of threads
- Experiment to find best options (some combinations can hurt performance)

```
#pragma acc kernels loop tile(2,2)
for(int x = 0; x < 4; x++){
  for(int y = 0; y < 4; y++){
    array[x][y]++;
  }
}
```



Source: https://www.olcf.ornl.gov/calendar/loop-optimizations-with-openacc/

PCAST Introduction

- Parallel Compiler Assisted Software Testing (PCAST) is a set of capabilities intended to help test for program correctness, and determine points of divergence
- Test changes to parts of a program, new compile-time flags, or ports to new compilers or new processors
- Also useful for detecting when results diverge between CPU and GPU versions of code
- Behavior is set through the **PCAST_COMPARE** environment variable, a comma-separated list of options that control various parameters of the comparison

Sources: https://www.pgroup.com/resources/pcast.htm
https://developer.nvidia.com/blog/detecting-divergence-using-pcast-to-compare-gpu-to-cpu-results/

autocompare

- Simplest way to invoke PCAST
- Use **-ta=tesla:autocompare** compiler flag to enable the autocompare feature
- Add **export PCAST_COMPARE=summary,abs=6,verboseautocompare** to **submit.sh** to get a summary of comparisons with absolute error tolerance up to 10^(-6) and verbose output
- When compiled with the **-ta=tesla:autocompare** flag, code in OpenACC compute regions will run redundantly on the CPU and GPU
- Any data in a **copy**, **copyout**, or **update host** directive will be compared when it is copied off the device.
- Note: **-ta=tesla:autocompare** implies **-ta=tesla:redundant** (which will be discussed later)

Source: https://www.pgroup.com/resources/pcast.htm

Functions & Directives: acc_compare

- Use the **acc_compare** function to explicitly compare data
- When called, it copies the data in GPU memory back to the host and compares it with the corresponding CPU memory. Data will be compared as many times as **acc_compare** is called, not just at the end of the data region.
- Contents of the data are kept in memory
- Must be called from CPU code, not from a device compute region (i.e. not inside acc parallel or acc kernels)
- Enter as **acc_compare(x, n)** where x is the data to compare and n is the number of elements to compare (not the the number of bytes)
- Can also enter as a directive with **#pragma acc compare(x[0:n])**
- Compile with **-ta=tesla:redundant** compiler option. This will redundantly run code in OpenACC compute regions on the GPU and CPU for comparison.

Source: https://www.pgroup.com/resources/pcast.htm

Functions & Directives: pgi_compare

- Similar to **acc_compare**, but data to be compared is written to a file
- Successive comparisons can be done in a quicker fashion since a "golden" copy is already on the disk
- Data file can grow very large depending on the amount of data the program is using and how often comparisons are done
- Use where the data involved is relatively small, or when it is necessary to compare results on different machines
- Enter as **pgi_compare(x, "type", n, "str", int)**, where x is the variable to be compared, "type" is a string of variable a's data type, n is the number of elements to be compared, “str” is the function name, and int is the line number. Can also enter as a directive with **#pragma pgi compare(x[0:n])**
- Running the program for the first time creates a **"pcast_compare.dat"**. Subsequent runs compare calculated data against this file. (Default filename can be changed with **PCAST_COMPARE** parameters.)

Source: https://www.pgroup.com/resources/pcast.htm

Breakout rooms

Navigate to the **GPU_workshop/Lesson_4_SharedMem/OpenACC/exercise** folder

Exercise: Matrix Multiplication with tile Clause

Add pragmas to **OpenACC/exercise** to make use of the **tile** clause.

Experiment with different tile dimension sizes to try to get the best speedup.

To build use **./build.sh**

To submit use **sbatch submit.sh**

Exercise: Matrix Multiplication with tile Clause Check

- Which tile dimension sizes gave you the best speedup?
- Were there dimension sizes that hurt performance (i.e. relative to using the collapse clause)?
- Were there dimension sizes that caused errors? Why?
- How did speedup compare to what was achieved in previous matrix multiplication exercises?

Extra Coding Practice: PCAST

Practice 1: autocompare

Update the **Makefile** and **submit.sh** in **PCAST_autocompare/exercise** to enable **autocompare.** Use it to verify that the code is running without errors.

Practice 2: Debug Scenario with acc_compare

In **PCAST_acc_compare/exercise**, we're curious about how much calculation will diverge between CPU and GPU. Add a call to **acc_compare()** in **matmul.cc.** Make appropriate changes to **Makefile** and **submit.sh** to compare results with different tolerances.

Practice 3: Debug Scenario with pgi_compare

A recent change has caused the gpuMatmul function in **PCAST_pgi_compare/exercise/matmul.cc** to fail verification. You have an existing **pcast_compare.dat** file from a previous successful run. Add a call to **pgi_compare()** beneath the **#pragma acc update host** and see if it helps you to find the problem. Make other necessary changes to **submit.sh** and/or the **Makefile** to enable **pgi_compare**.

You can find a full list of options for the **PCAST_COMPARE** environment variable here: https://www.pgroup.com/resources/pcast.htm

Practice CUDA/OpenACC Exercise

- Vector dot product using shared memory

Nsight Exercise

To get more useful profiling data, you can use the following command:

nsys profile -t nvtx,[cuda or openacc] --stats=true --force-overwrite true -o profileData ./output_name.exe

References

1. Using Tigervnc.
https://www2.cisl.ucar.edu/resources/computational-systems/casper/using-remote-desktops-casper-vnc

2. Using Nsight profiler.
https://www.youtube.com/watch?v=kKANP0kL_hk

3. Nsight command flags and descriptions.
https://docs.nvidia.com/nsight-systems/UserGuide/index.html