TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Introduction to CUDA (Part I)

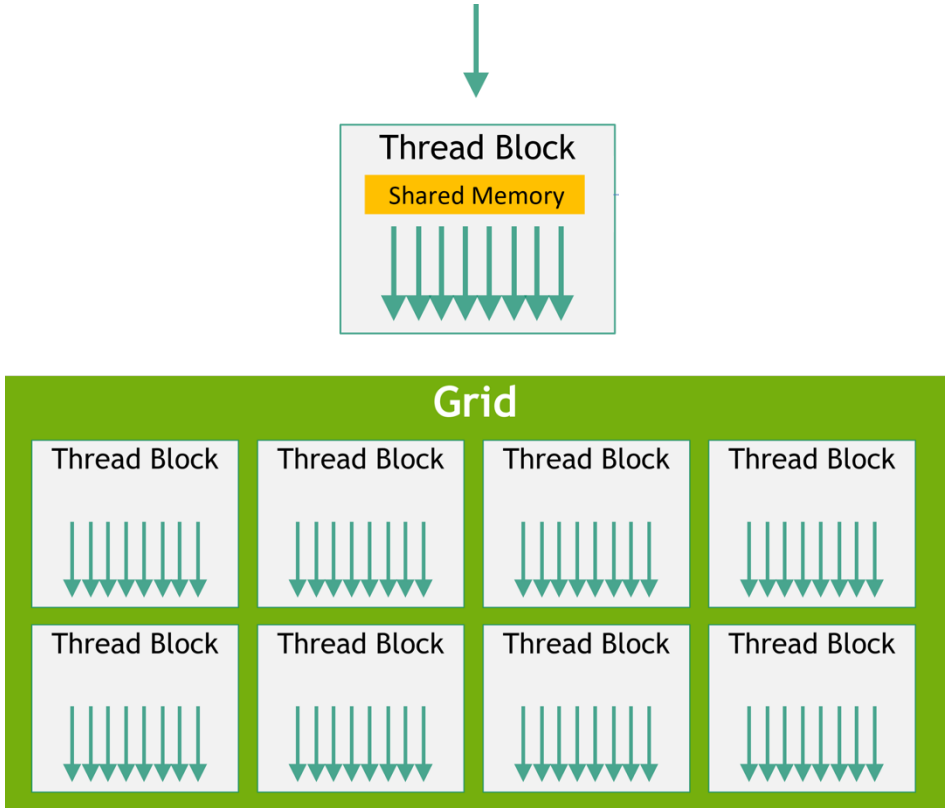
Applied Parallel Programming
Fall 2025

by Zuzanna Jedlinska
zuzanna@tacc.utexas.edu

Slides https://tinyurl.com/tacc25-hpc-vista

What is CUDA

- Parallel programming platform for NVIDIA GPUs
- Thread-based programming model
- Available as language extensions (basic language + CUDA)
- Software developed to match the hardware



Why use a GPU?

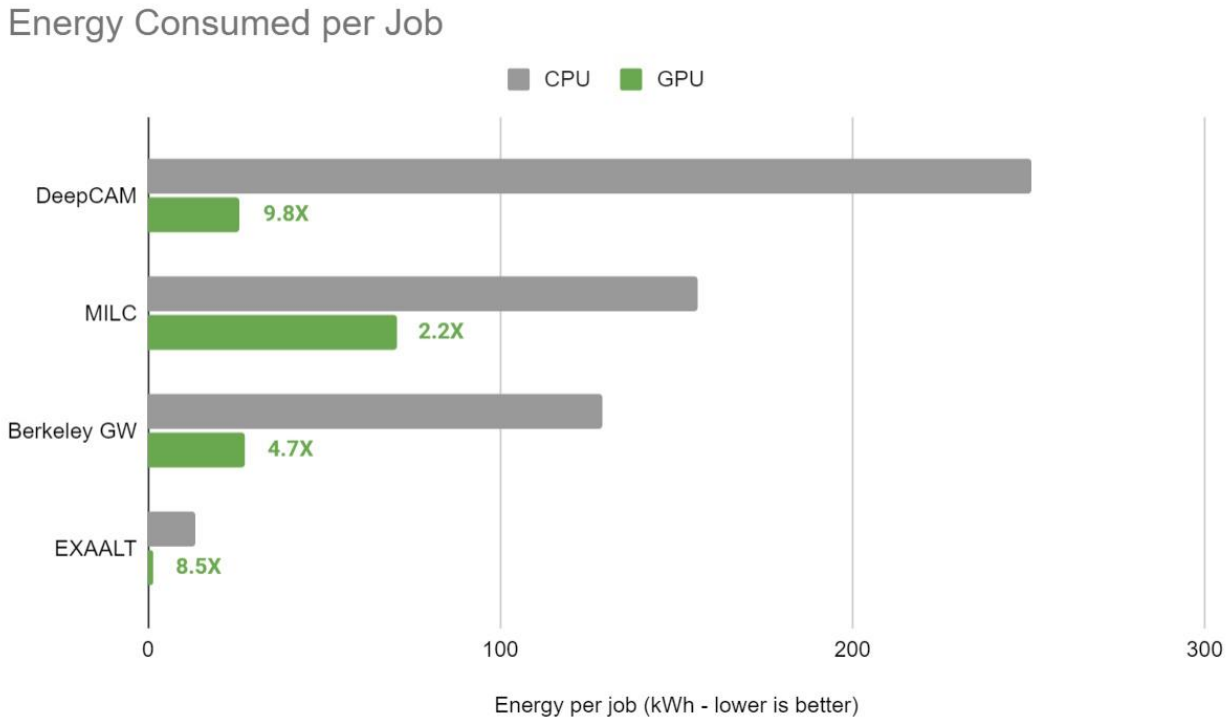
In HPC we want efficiency and performance

CPU limiting factors:

- (Mostly) Serial execution
- Moore's law limit
- Clock speed

Benefits of using GPUs:

- Massive parallelism
- Energy efficiency



https://blogs.nvidia.com/blog/gpu-energy-efficiency-nersc/

GPUs in scientific computing

- Machine Learning
- Molecular Dynamics
- Fluid Dynamics
- Quantum Mechanics
- Scientific Visualization

Not a silver bullet

What can be offloaded to a GPU

- Computation intensive tasks (high compute to data load ratio)
- Data accessed in large continuous chunks
- Matrix operations

What should not

- Code with a lot of branching instructions
- Lots of interdependence between threads
- Processing done on small batches of data
- Irregular and scattered data access pattern

Ways to program a GPU

GPU programming “languages”:

- CUDA for NVIDIA (C/C++ and Fortran)
- HIP for AMD (C only)

Pragma (compiler directive) based device offloading:

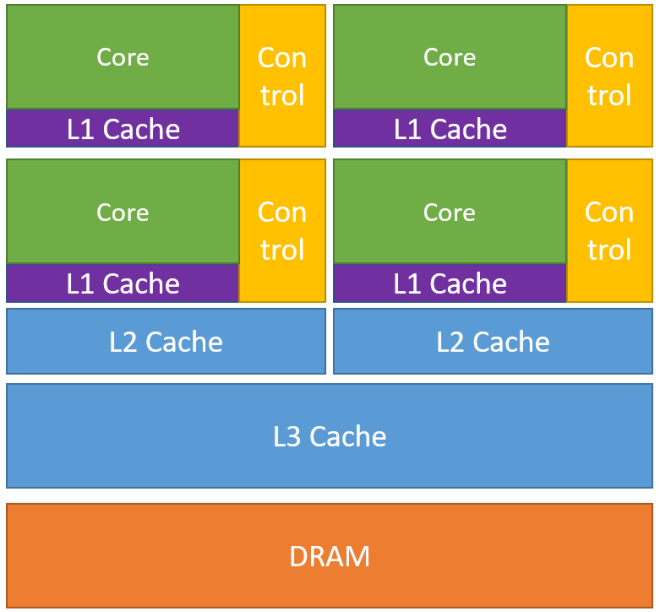
- OpenMP
- OpenACC

What's in the CUDA toolbox

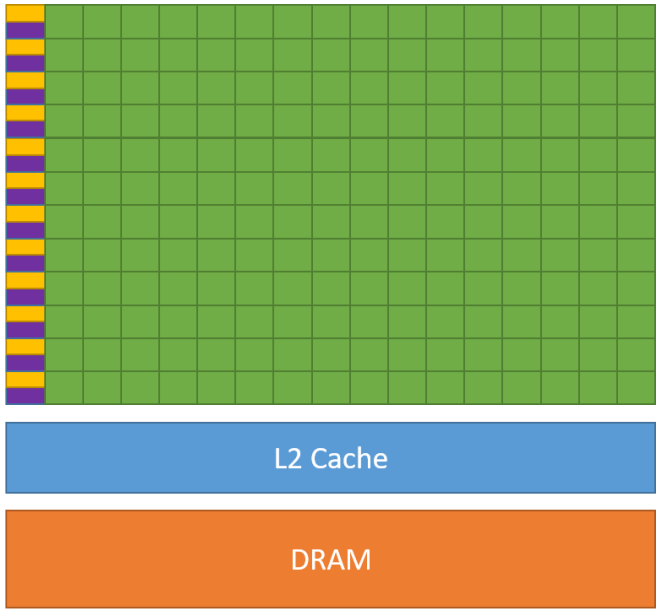
- CUDA: C/C++ and Fortran
- Specialized libraries:
  - Math: cuFFT, cuBLAS
  - Algorithms: Thrust
  - Deep Learning: cuDNN
- Compilers: nvcc, nvc, nvc++, nvfortran
- CUDA-aware MPI (OpenMPI)
- Profiling tools: Nsight Systems, Nsight Compute

CPU vs GPU

CPU	GPU
Low core count	Very high (1,000s) core count
Low latency	High latency but high throughput
Good for serial execution, can be parallel	Optimized for parallel execution
Fast context switching, branching	Good for SIMD, avoid branching

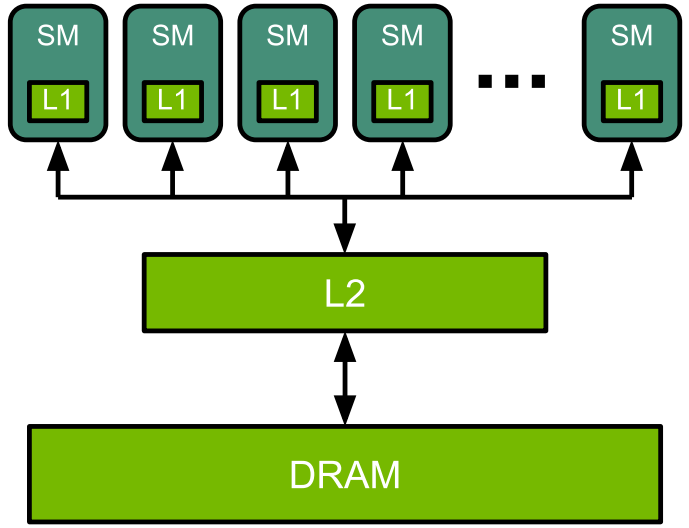


CPU



GPU

Multi-level of (NVIDIA) GPU organization



Physical Components
GPU card
Streaming Multiprocessors
Streaming Processors
CUDA cores / Tensor Cores

Memory
Global Memory (DRAM)
L2 Data Cache
L1 Data Caches
Registers

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#gpu-arch

Streaming Mutiprocessor (SM)

Memory in SM:

- L1 Data Cache / Shared Memory

- Divided into Streaming Processors (SP)
  - ➤ CUDA cores and Tensor core
  - ➤ Registers
  - ➤ Load/Store Units
  - ➤ Special Function Unit (cos(x), sin(x), e^x)

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

What different numbers mean

Compute Capability (CC) - defines what features and instructions are available for a specific GPU architecture.

See CUDA Programming guide 17.2. Features and Technical Specifications (Tables 27 & 28)

- Higher CC means more features.
- Tesla (CC 1.x) → Hopper (9.0) → Blackwell 12.0
- Example: Tensor cores available from CC 7.0+, thread block clusters 9.0+*CUDA*

CUDA Toolkit (12.9.0 as of May 2025)

Feature	A100 80GB SXM	H100 80GB SXM
GPU Architecture	Ampere	Hopper
GPU Memory	80GB HBM2e	80GB HBM3
Memory Bandwidth	2039 GB/s	3352 GB/s
L2 Cache	40MB	50MB
FP64 Performance	9.7 TFLOPS	33.5 TFLOPS
FP32 Performance	19.5 TFLOPS	66.9 TFLOPS
RT Cores	N/A	N/A
TF32 Tensor Core	312 TFLOPS	989 TFLOPS
FP16/BF16 Tensor Core	624 TFLOPS	1979 TFLOPS
FP8 Tensor Core	N/A	3958 TFLOPS
INT8 Tensor Core	1248 TOPS	3958 TOPS

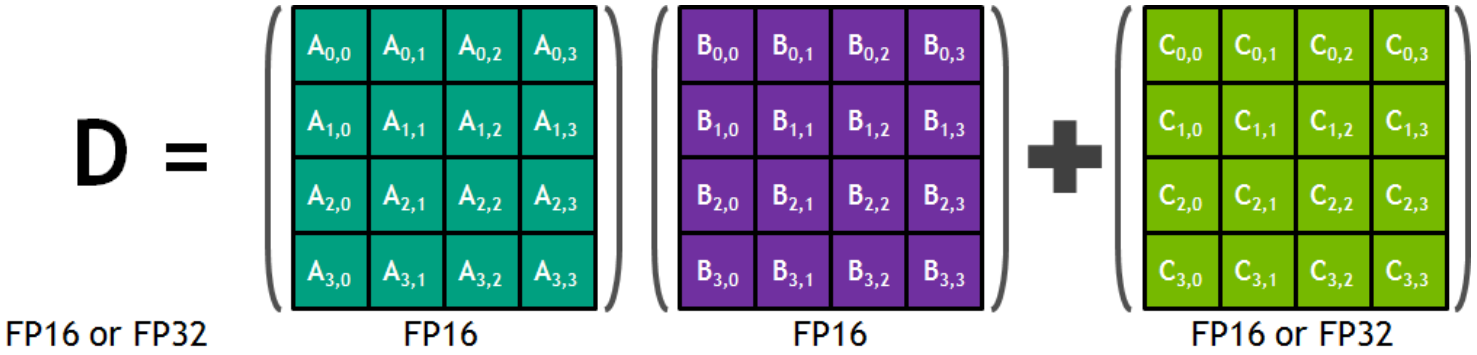
https://www.horizoniq.com/blog/h100-vs-a100-vs-l40s/

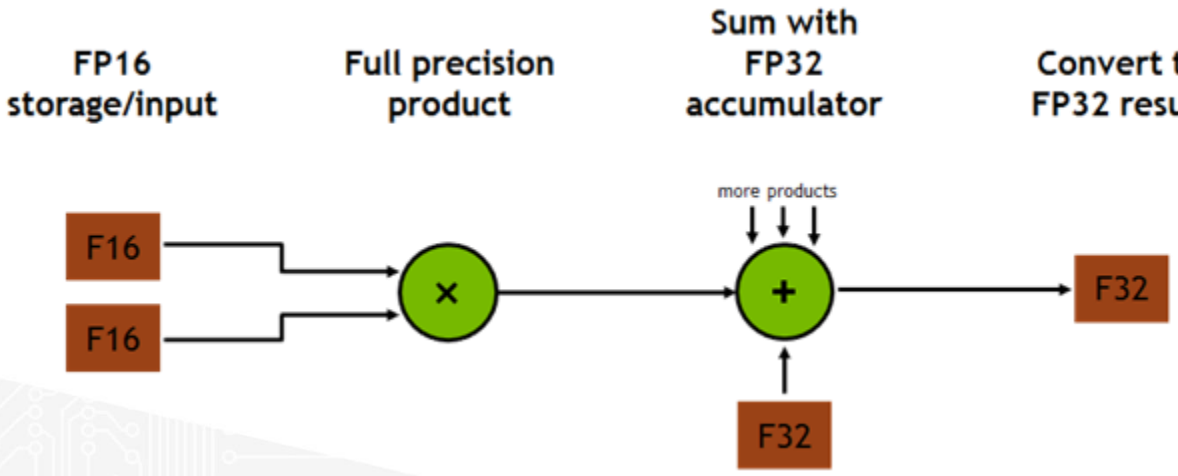
Tensor cores

Specialized cores operate on 4x4 matrices

Performed matrix multiply and accumulate

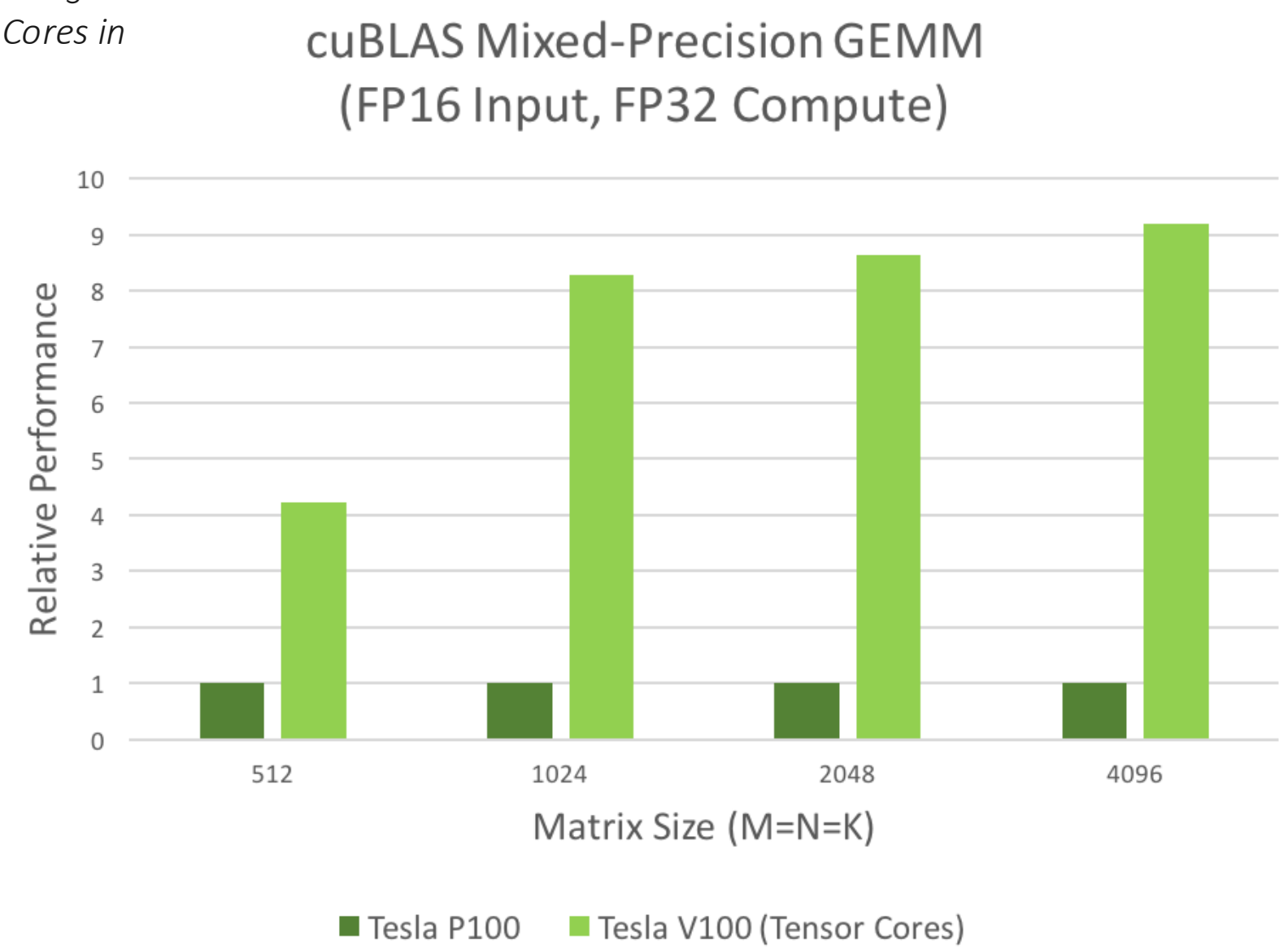
Used by



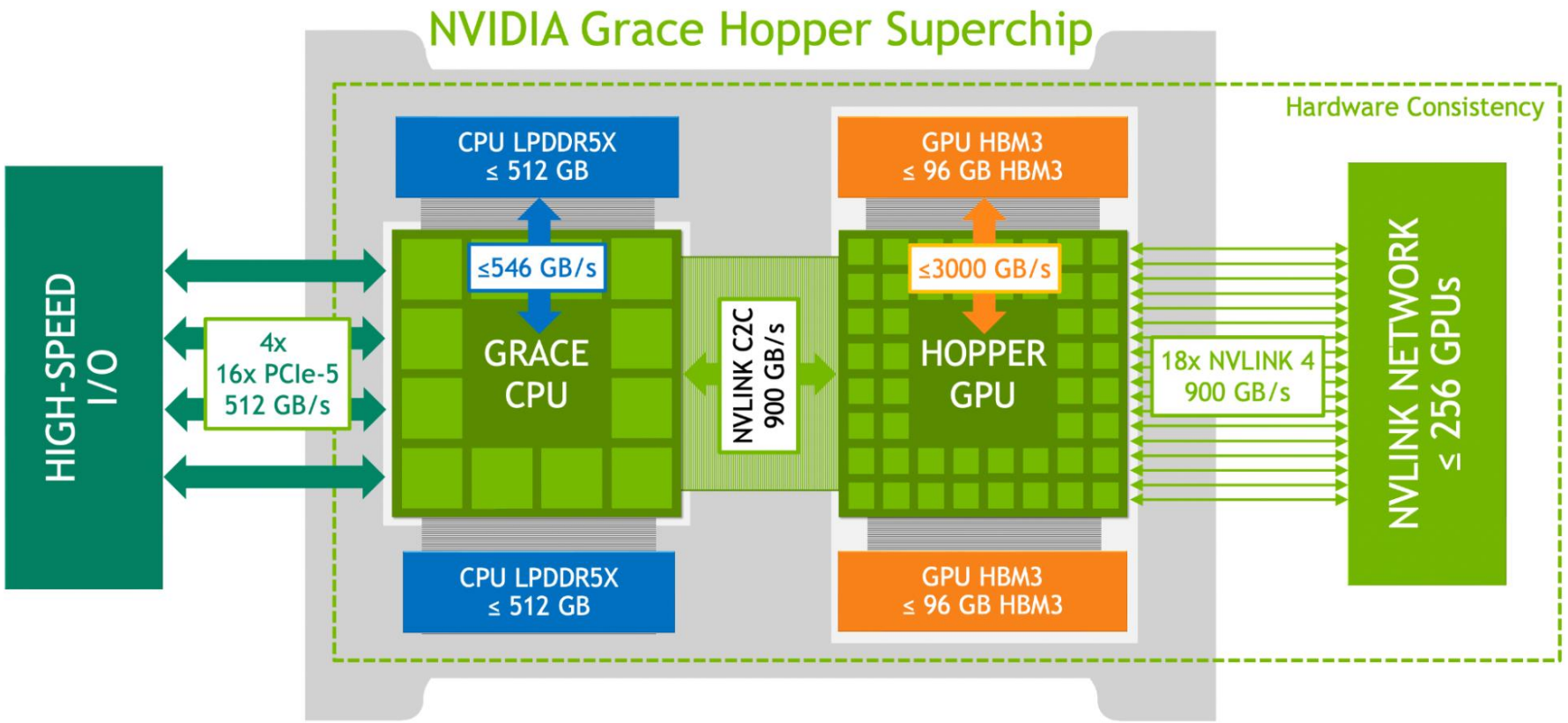


From Nvidia Technical Blog Programming Tensor Cores in CUDA 9

From Nvidia Technical Blog Programming Tensor Cores in CUDA 9

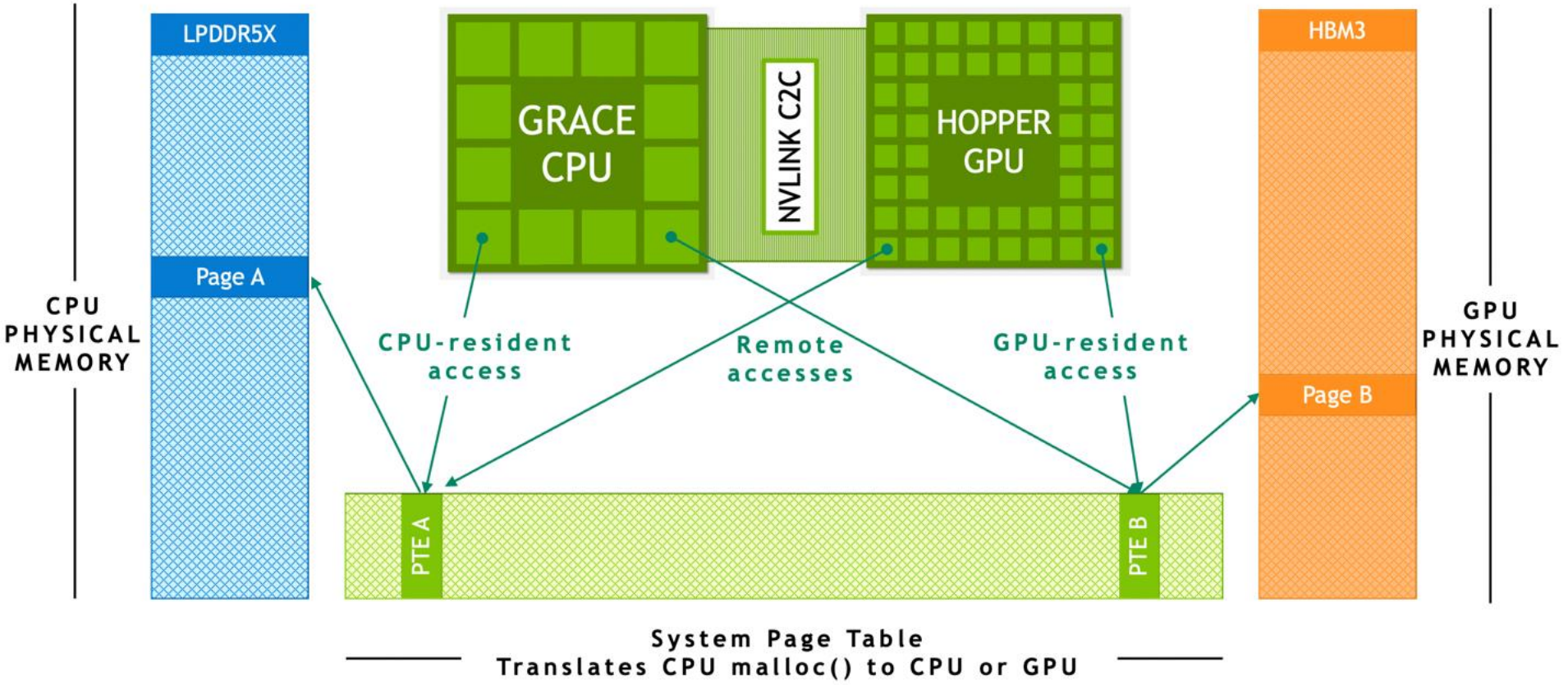


Grace-Hopper Superchip



From Nvidia Technical Blog

Unified Memory + Heterogeneous Memory Management (HMM)



From Nvidia Technical Blog

Log into vista

Login to vista

ssh **your_user_name**@vista.tacc.utexas.edu

Request interactive session gh, type number corresponding to reservation (number 2 for the example on the right), ENTER

idev –p gh

```
=> Select RESERVATION,  options are:
No  Res_Name   Account  Partition
 2  Fall-HPC   TRA24006    gh
```

(or full syntax if above fails)

idev -p gh -r PerfCounter_testing –A TRA24006

Copy tutorial files to your scratch directory and unpack.

Go to the folder 0_Query in the unpacked directory.

cds && cp ~train00/IntroToCUDA.tar .

/home1/00692/train00/IntroToCUDA.tar

tar -xvf IntroToCUDA

cd IntroToCUDA/0_Query

Compile the source code and run the executable

nvcc –o query query.cu

./query

Query the device properties

In the *0_Query directory, execute ./query*

```
c613-091[gh](1057)$ ./query

 * You have 1 CUDA-enabled device *

=== Information for device number 1 ===
 Device name: NVIDIA GH200 120GB
 Compute Capability: 9.0
 Maximum threads per SM:  2048
 Maximum threads per block (blockSize): 1024
 Max thread block size in x:1024, y:1024, z:64
 Max grid size in x:2147483647, y:65535, z:65535
 Total Global Memory: 95.000000 GB
 Peak Memory Bandwidth:  4022.784000 GB/s
```

```
int numOfDevices;

cudaGetDeviceCount(&numOfDevices);

printf("\n\n * You have %d CUDA-enabled devices *\n\n",numOfDevices);

//enumerate though devices
for (int i = 0; i < numOfDevices; i++) {

cudaDeviceProp properties;
cudaGetDeviceProperties(&properties, i); //puts all properties in the properties variable
printf(" === Information for device number %d ===\n", numOfDevices);

//Properties can be accessed by their name

printf(" Device name: %s\n", properties.name);
printf(" Compute Capability: %d.%d\n", properties.major, properties.minor);
printf(" Maximum threads per SM: %d\n", properties.maxThreadsPerMultiProcessor);

}
```

Using nvidia-smi and property query

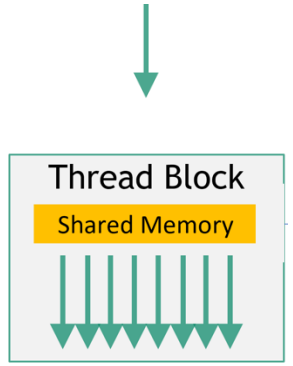
```
c610-071[gh](1009)$ nvidia-smi
Tue Jun  3 15:58:59 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 560.35.03              Driver Version: 560.35.03      CUDA Version: 12.6     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GH200 120GB             On  |   00000009:01:00.0 Off |                    0 |
| N/A   29C    P0             73W /  900W |       1MiB /  97871MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

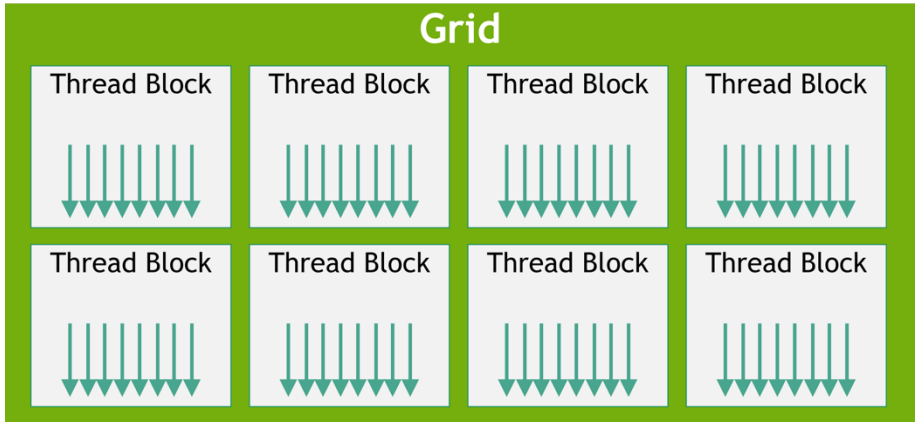
Execution Model & Hardware Mapping

Threads in a warp execute in-step

Only guarantee of warp concurrency

- Threads are mapped to CUDA cores
- Blocks mapped to SMs
- Grid mapped to the GPU





Memory Hierarchy

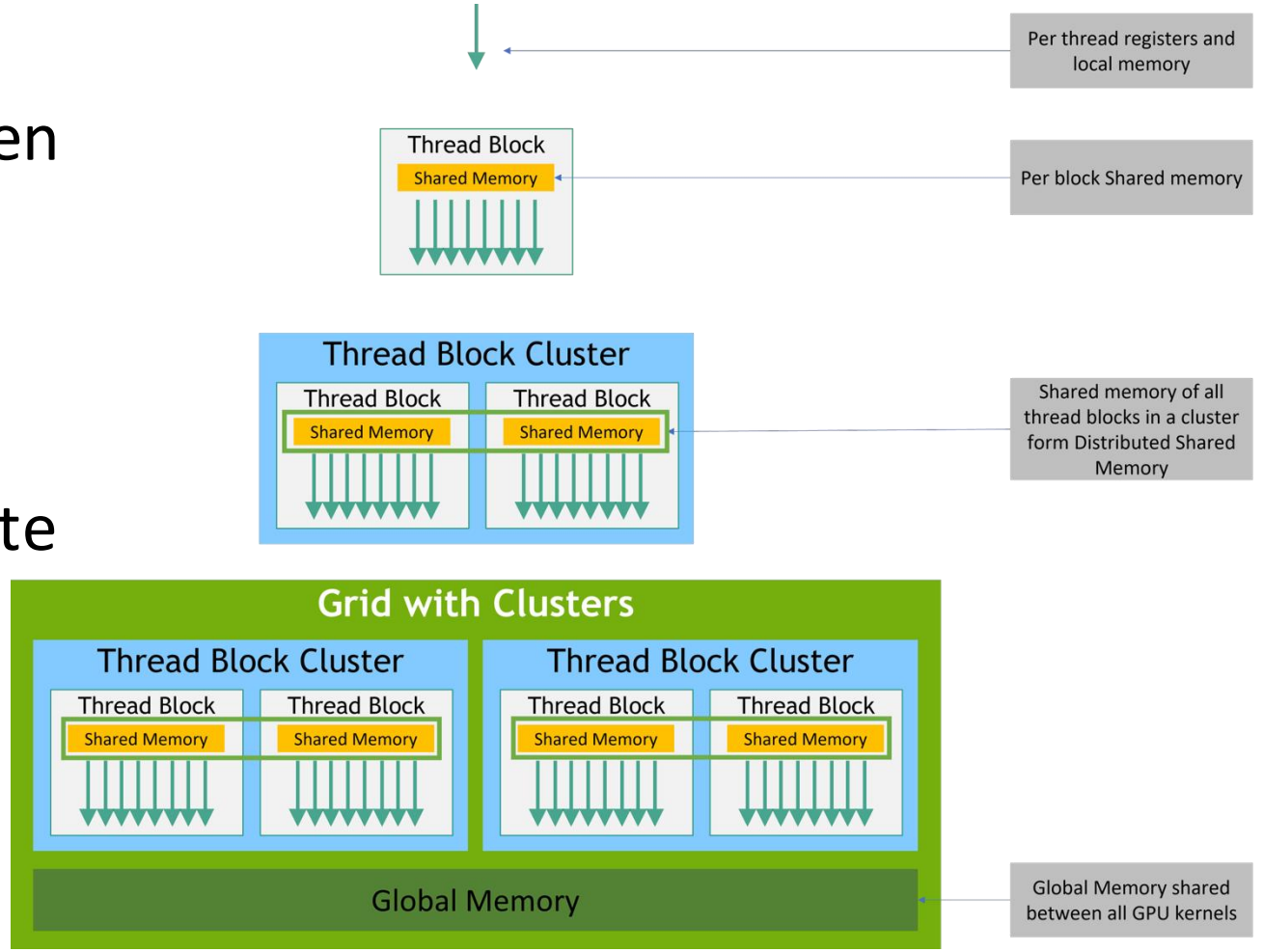
High latency in data transfer hidden with multiple levels of memory

Registers

L1 local cache / Shared Memory

L2 shared by all SMs – intermediate in Host communication

Global Memory (DRAM)



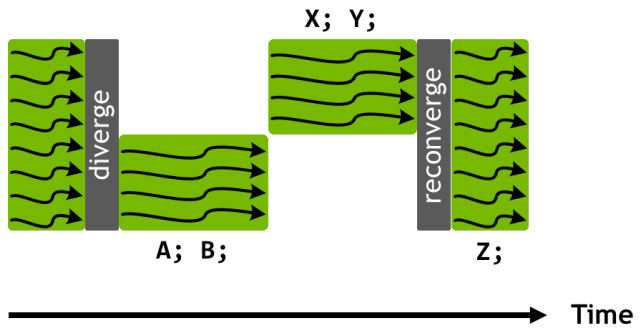
Warps and SIMT Execution Pipeline

Threads in a warp execute in-step

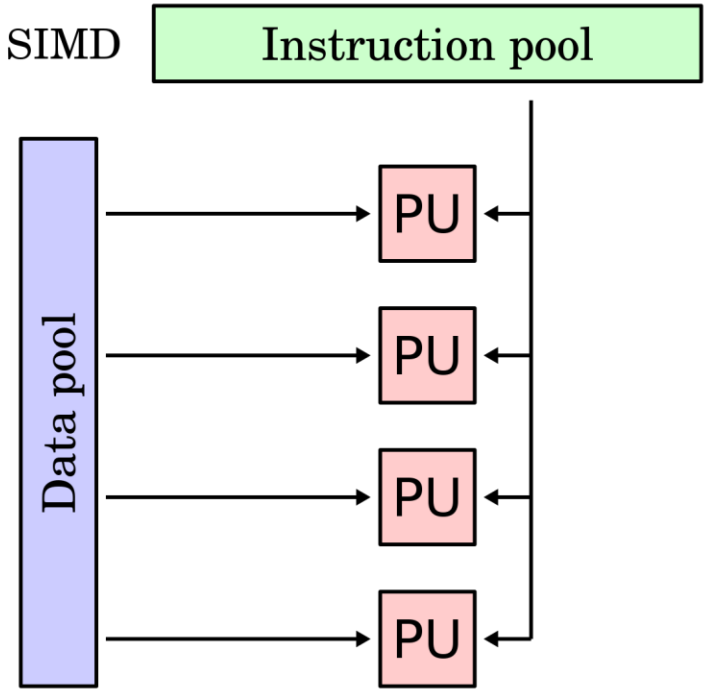
Only guarantee of warp concurrency

Divergent branches (if; else) should be avoided

```
if (groupThreadID.x < 4){
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



From Nvidia Technical Blog



PU – Processing Unit, here a GPU thread

https://en.algorithmica.org/hpc/simd/

Migrating the data

Data is initialized on the CPU, then copied to the GPU

CPU ←→ GPU transfers are slow, keep them to the minimum

Ways to transfer the data:

- Allocate space on the GPU, copy from CPU to GPU (and back)
- Use Unified (Managed) Memory

Data allocation and movement

```
allocate memory on the device
cudaMalloc(device_pointer,  size)

copy data between host and device
cudaMemcpy(dest_ptr,  src_ptr, size, direction)
```

Data copy directions

0: cudaMemcpyHostToHost
1: cudaMemcpyHostToDevice
2: cudaMemcpyDeviceToHost
3: cudaMemcpyDeviceToDevice

```
int *array, *d_array;

cudaMalloc(&d_array, N * sizeof(int));

cudaMemcpy(d_array, array, N*sizeof(int), cudaMemcpyHostToDevice);
```

Using Unified Memory

cudaMallocManaged(void** ptr, size_t size);

- Pointer accessible both from the host and device
- Software handles data migration, data migrated as needed
- Simplicity but less control

Kernel – a GPU function

Kernels declared as normal functions with ***__global__*** attribute

__global__ *myKernel*(kernel parameters){ …kernel body…};

__global__ makes the function callable from host and device

Called with information about # of blocks and threads/block

myKernel<<< ***# Blocks*** , ***Threads per block*** >>>(parameters);

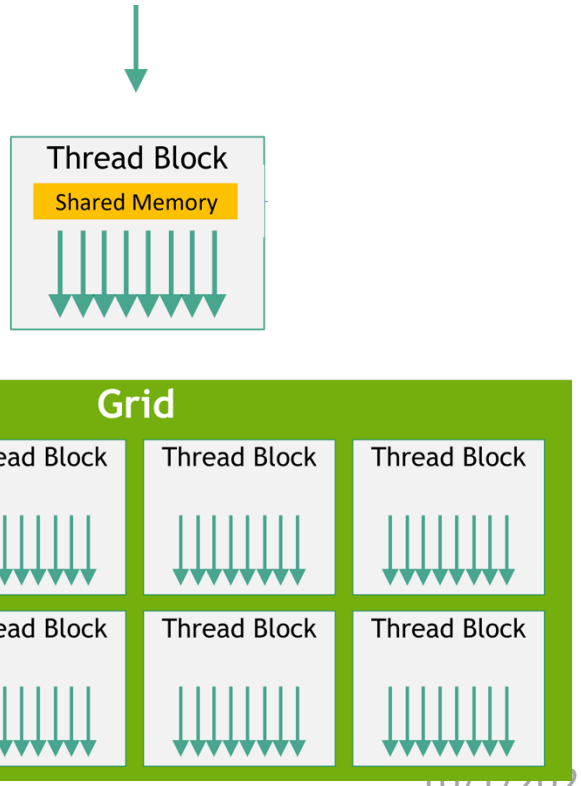
Kernel configuration

Chevron syntax <<< >>> specifies the resources provided to the kernel

Two required positional parameters <<<**GridSize**, **BlockSize** >>>

GridSize = Number of thread blocks

BlockSize = Number of threads in a block



Kernel configuration

Two-tier hierarchy:

- **Grid** of blocks
- **Block** of threads

Why the division?

- Entire block assigned to one SM – shared memory, synchronization
- Grid specifies how many blocks we want to launch
- Since block is mapped to SM, it will only start when resources are available

Gridsize and blocksize

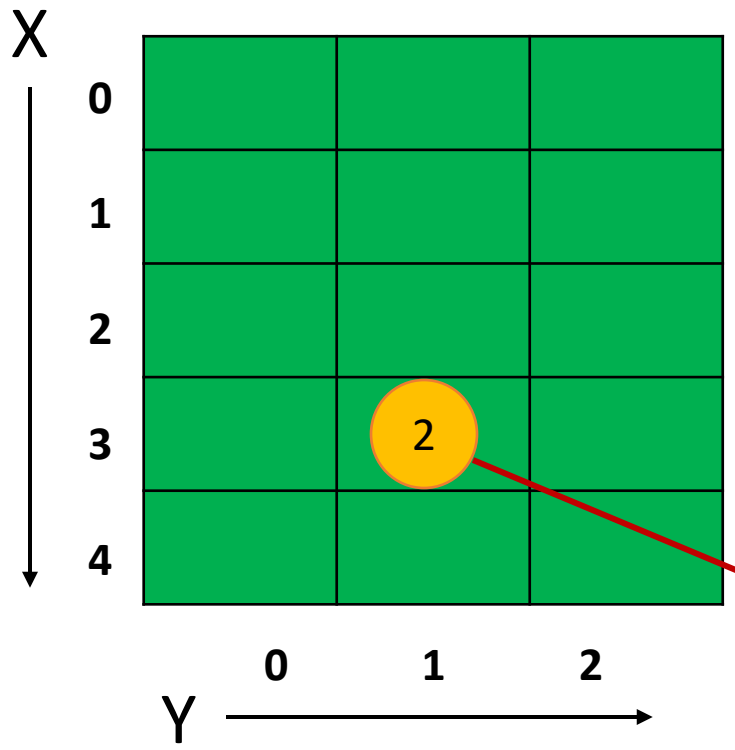
Grid size and block size are dim3 (x, y, z): can be 1D, 2D or 3D

Block size limited by max number of threads 1024. Block size must be the multiple of 32 (warp size).

Grid size imposes limit in each direction, no limit for total grid size (x: 2^{31}-1, y: 65535, z: 65535)

Multidimensional grid helps to map to multidimensional data

Indexing the threads




```
dim3 GridSize(5,3) dim3 BlockSize(512)
someKernel<<<GridSize,BlockSize>>>();
```

Position on the grid = GridSize.x *GridIdx.y + BlockId.x → 5 x 1 + 3 = 8

Position in the block = ThreadIdx.x

Global Id = (GridSize.x *GridIdx.y + BlockId.x) *BlockSize.x + ThreadIdx.x

Converting CPU code to GPU code

A simple serial CPU function to square an array

```
void squareArray(int* array, int N){

for (int i=0; i < N; i++){
      array[i] = array[i] * array[i];
   }
}
```

Mapping loop iterations to threads

```
void squareArray(int* array, int N){

for (int i=0; i < N; i++){
        array[i] = array[i] * array[i];
    }
}
```

```
__global__ d_squareArray(int* d_array, int N){

    int tid = BloxkSize.x * BlockIdx.x + ThreadIdx.x;

    // don't go out of bounds
    if (tid < N){
    // each thread maps to a loop iteration
    d_array[i] = d_array[i] * d_array[i];
    }
}
```

Data allocation and movement

```
//host allocation and initialization

const int N = 1024;

int *array = (int *)malloc(sizeof(int) * N);

for (int i = 0; i < N; i++) {array[i] = i;}

//device allocation and copy

int *d_array;

cudaMalloc(&d_array, N * sizeof(float));

cudaMemcpy(d_array, array, N*sizeof(int), cudaMemcpyHostToDevice);
```

Running the function / kernel

```
// CPU
squareArray(array, N);

int blockSize = 512;
int gridSize = N/blockSize;

//GPU
d_squareArray <<<gridSize, blockSize>>>(d_array, N);
cudaMemcpy(array, d_array, N*sizeof(int), cudaMemcpyDeviceToHost);

//deallocate the memory
free(array);
cudaFree(d_array);
```

Lab 1

First GPU kernel

Lab Outline

1. Log in to the cluster (vista) and request the gh node
2. Analyze and compile the code
3. Run the first kernel
4. Measure code performance
5. Use profiling tools

System setup

Login to vista

ssh **your_user_name**@vista.tacc.utexas.edu

Request interactive session gh, type number corresponding to reservation (number 2 for the example on the right), ENTER

idev –p gh

(or full syntax if above fails)

idev -p gh -r PerfCounter_testing –A TRA24006

(If you haven't already) Copy the tutorial files to your scratch directory and unpack. Go to 1_FirstKernel in the unpacked directory.

cds && cp ~train00/IntroToCUDA.tar .

tar -xvf IntroToCUDA

cd IntroToCUDA/1_FirstKernel

Compile the source code (arrayAdd.cu) with nvcc, create output file arrayAdd and run it

nvcc –o arrayAdd arrayAdd.cu

./arrayAdd

=> Select RESERVATION, options are:

No	Res_Name	Account	Partition
2	Fall-HPC	TRA24006	gh

Kernel structure

```
__global__ void arrayAdd(float *d_a, float *d_b, float *d_out, int size)
{
    // Calculate the global thread ID

    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // Make sure we don't go out of bounds

    if (tid < size) {
       d_out[tid] = d_a[tid] + d_b[tid];
       }
}
```

Check for errors

```
cudaError_t err;

err = cudaGetLastError();
if (err != cudaSuccess)
{
   printf("CUDA Error:%s\n",cudaGetErrorString(err));
   return -1;
}
```

CUDA events

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);

… GPU code to time …

cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
```

More precise than CPU timers for measuring GPU execution time

cudaEventRecord puts a time stamp into a given event

cudaEventElapsedTime measures time between the events in ms.

Sample output

Effective bandwidth: (total bytes read or written)/execution time

Notice the warm-up run before the actual timing

```
./arrayAdd
Copy H->D took 0.144992 ms
CUDA kernel launch with 40 blocks of 256 threads
Kernel warm up time: 36.640991 ms
Matrix addition took 0.004992 ms
Effective Bandwidth (GB/s): 24.038462
Copy D->H took 0.015424 ms
```

Profiling tools

For more insight we can use profiling:

Nsight Systems – for profiling entire code
Nsight Compute – profiling specific kernel, detailed report

Repots can be viewed in GUI on any system

https://developer.nvidia.com/nsight-systems/get-started
https://developer.nvidia.com/nsight-compute

Using Nsight Systems

```
nsys profile  -o arrayAddprof --stats=true ./arrayAdd
```

The command will profile the code, create arrayAddprof.nsys-rep report file and print statistics to the terminal.

To display them again from the report, use:

```
nsys stats arrayAddprof.nsys-rep
```

Example output

```
[6/8] Executing 'cuda_gpu_kern_sum' stats report

 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)             Name
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  ----------------------------
    100.0            3,360          2   1,680.0   1,680.0     1,664     1,696         22.6  arrayAdd(int *, int *, int *, int)

[7/8] Executing 'cuda_gpu_mem_time_sum' stats report

 Time (%)  Total Time (ns)  Count  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)          Operation
 --------  ---------------  -----  --------  --------  --------  --------  -----------  ---------------------------
     70.1            6,080      2   3,040.0   3,040.0     2,912     3,168        181.0  [CUDA memcpy Host-to-Device]
     29.9            2,592      1   2,592.0   2,592.0     2,592     2,592          0.0  [CUDA memcpy Device-to-Host]

[8/8] Executing 'cuda_gpu_mem_size_sum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)          Operation
 ----------  -----  --------  --------  --------  --------  -----------  ---------------------------
      0.080      2     0.040     0.040     0.040     0.040        0.000  [CUDA memcpy Host-to-Device]
      0.040      1     0.040     0.040     0.040     0.040        0.000  [CUDA memcpy Device-to-Host]
```

Using Nsigh Compute

Command below tells Nsight Compute to profile arrayAdd kernel in the executable arrayAdd. Report will be saved in **arrayAddprof.ncu-rep.** Report name is specified with *–o* option and the .ncu-rep extension added automatically.

--set option specifies which metrics to collect.

```
ncu -o arrayAddprof --set full --kernel-name=arrayAdd ./arrayAdd
```

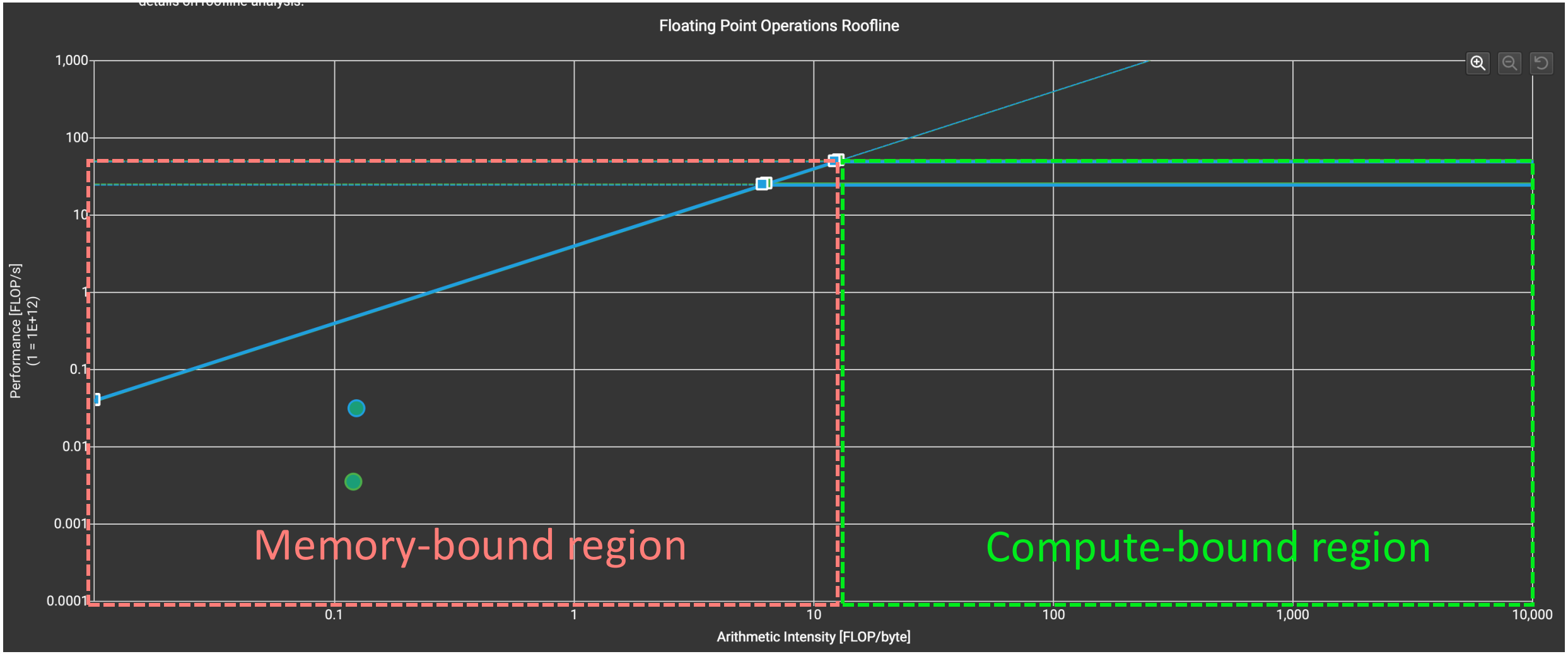
Use --import mode to display report details in the terminal

```
ncu --import arrayAddprof.ncu-rep
```

Section: Memory Workload Analysis

Metric Name	Metric Unit	Metric Value
Memory Throughput	Gbyte/s	25.17
Mem Busy	%	0.66
Max Bandwidth	%	0.63
L1/TEX Hit Rate	%	0
L2 Compression Success Rate	%	0
L2 Compression Ratio		0
L2 Hit Rate	%	36.81
Mem Pipes Busy	%	1.06

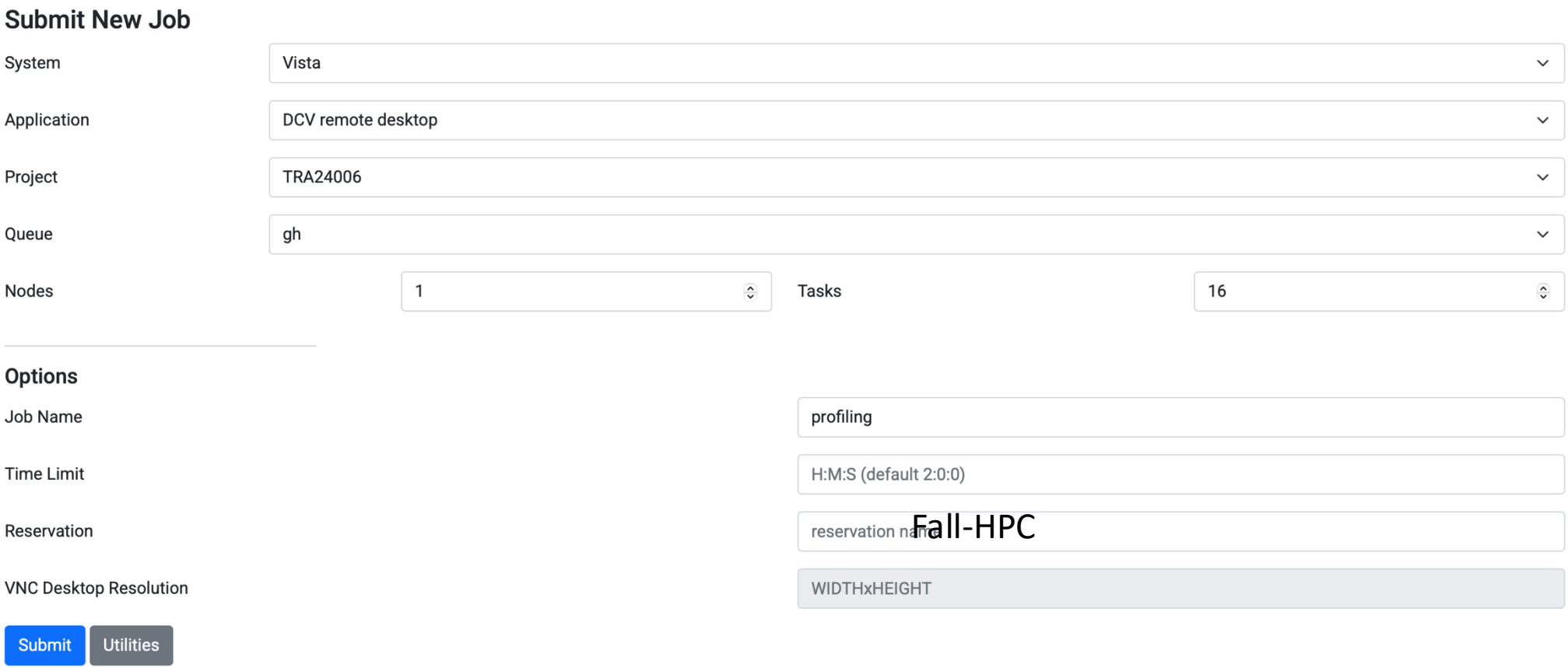
```
Section: GPU Speed Of Light Throughput
---------------------- ----------- ------------
Metric Name            Metric Unit Metric Value
---------------------- ----------- ------------
DRAM Frequency                 Ghz        2.58
SM Frequency                   Ghz        1.51
Elapsed Cycles               cycle       4,298
Memory Throughput                %        0.77
DRAM Throughput                  %        0.73
Duration                        us        2.85
L1/TEX Cache Throughput          %        2.14
L2 Cache Throughput              %        1.50
SM Active Cycles             cycle      662.30
Compute (SM) Throughput          %        1.09
---------------------- ----------- ------------
```


Floating Point Operations Roofline
Performance [FLOP/s]
(1 = 1E+12)
1,000
100
10
1
0.1
0.01
0.001
0.0001
0.1
1
10
100
1,000
10,000
Arithmetic Intensity [FLOP/byte]
Memory-bound region
Compute-bound region

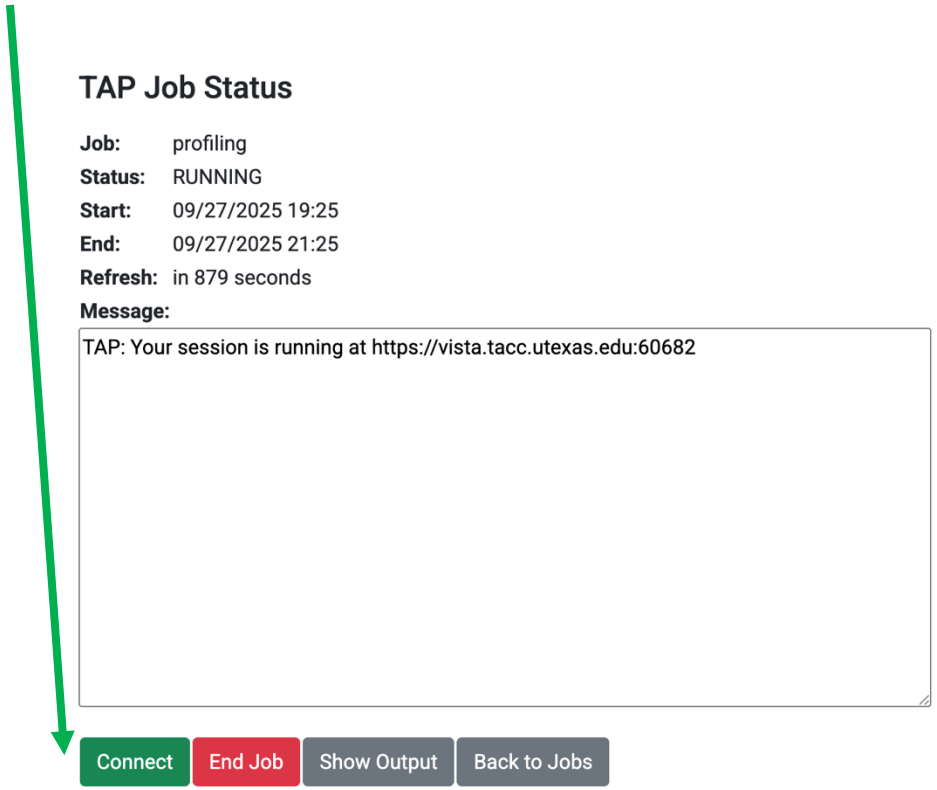
Using the GUI

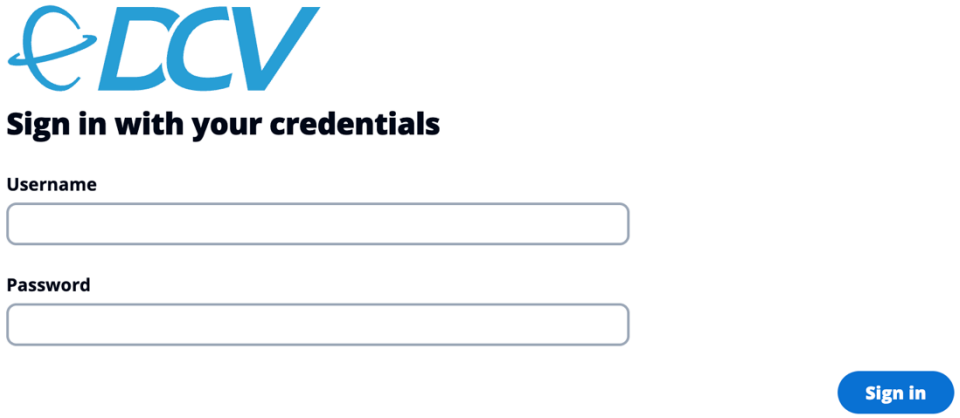
We will use TACC Analysis Portal (TAP) to get remote desktop on the machine
Go to https://tap.tacc.utexas.edu and log-in with TACC credentials

Don’t use Safari browser



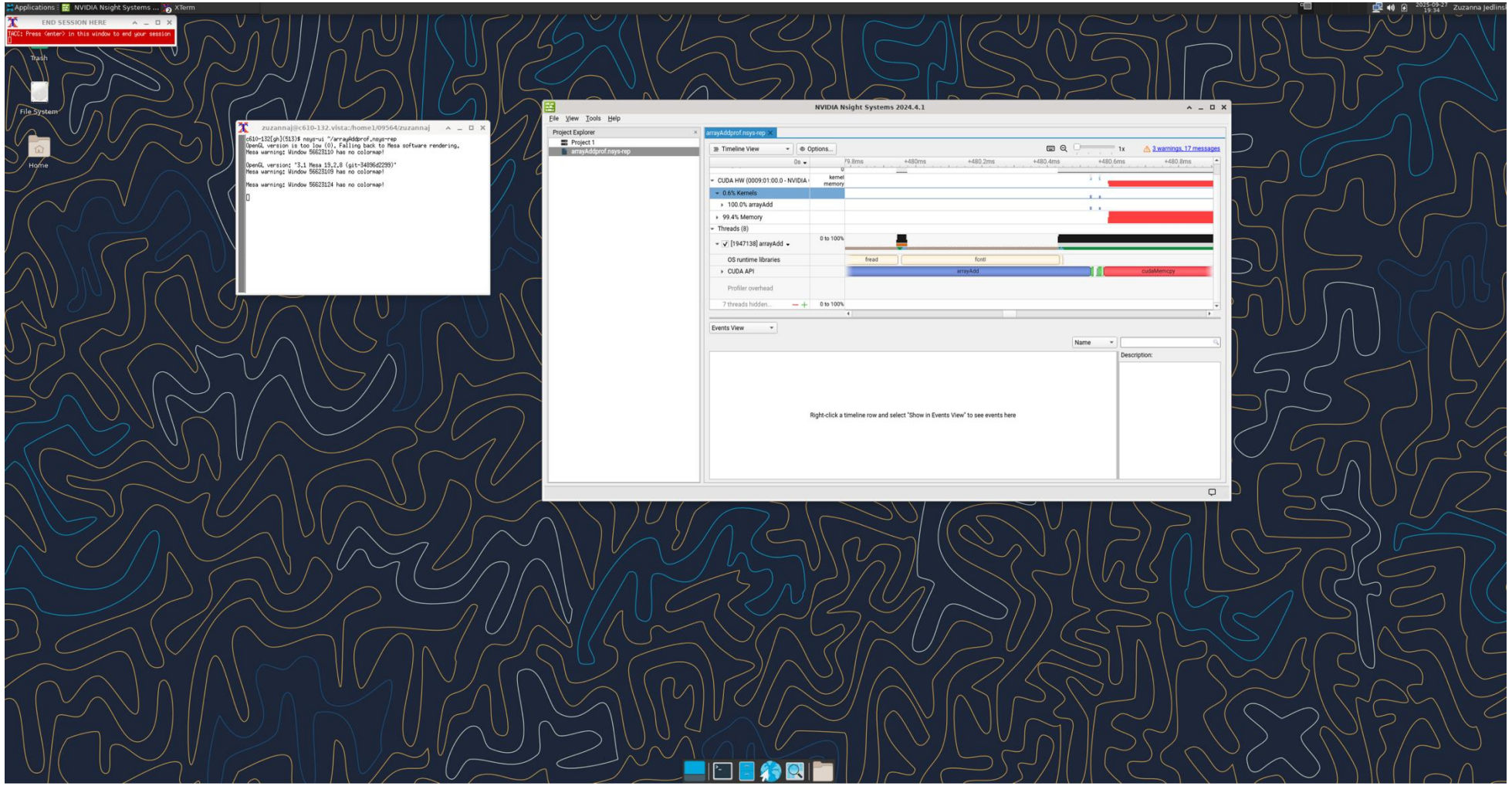
Connect and credentials again – you will get a remote desktop.





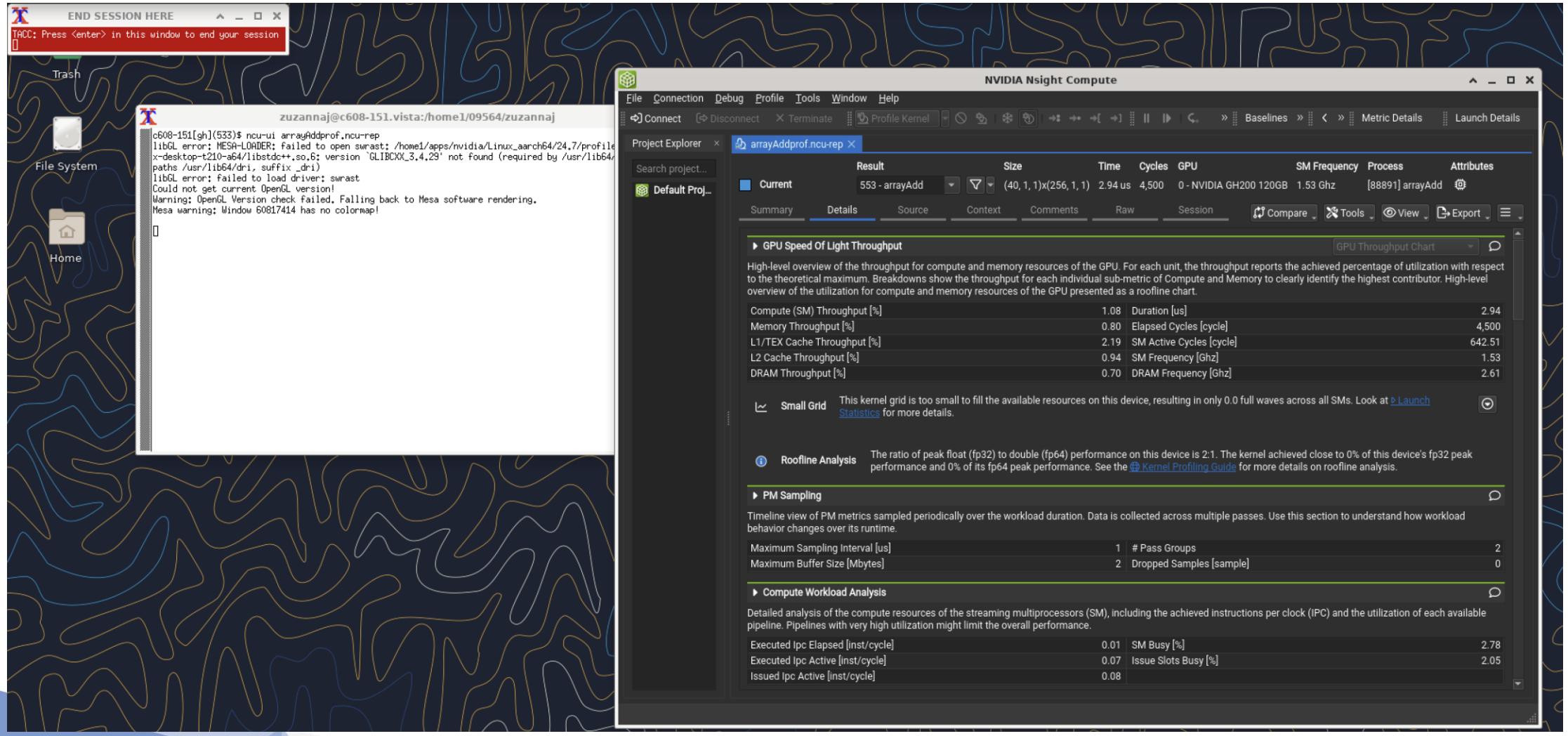
cds && cd IntroToCUDA/1_FirstKernel

nsys-ui ~/arrayAddprof.nsys-rep



cds && cd IntroToCUDA/1_FirstKernel

ncu-ui ~/arrayAddprof.ncu-rep



Things to try

1. Try different array sizes and kernel configurations (blocks per grid and threads per block). How does it affect the execution time?
2. Compare the data copy time to the kernel execution time for different array sizes.
3. Generate profiler report for different array sizes/kernel configuration and examine them.

TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU

The University of Texas at Austin

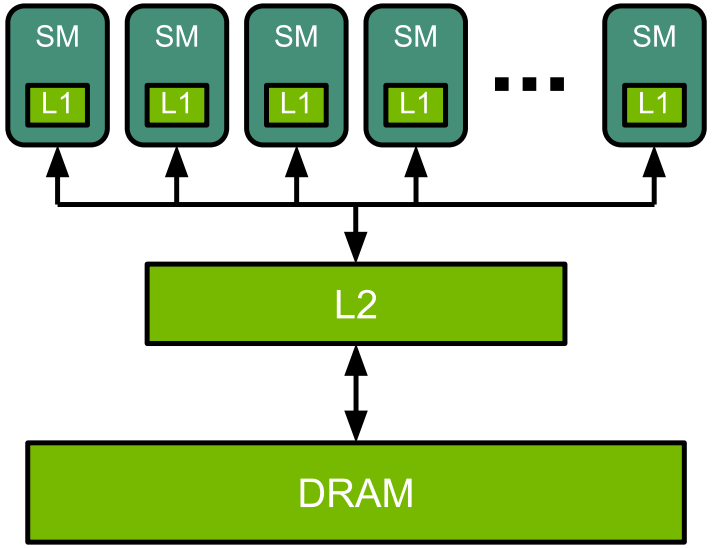
Introduction to CUDA II

Performance and optimization

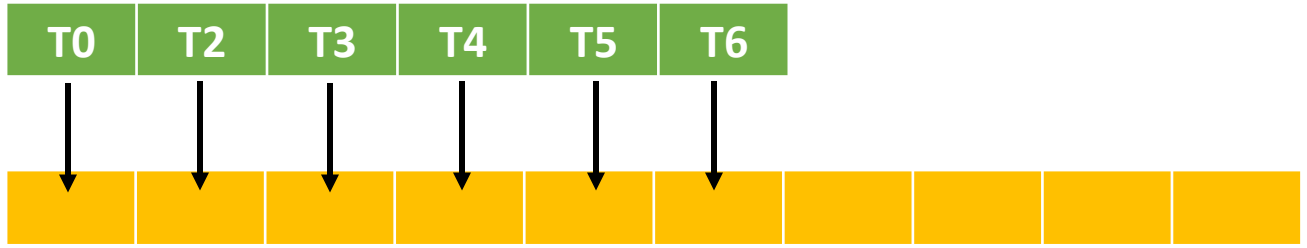
Review on memory types

Location	Capacity	Bandwidth (GB/s)	Latency (ns)
L1 cache/shared memory	192 KB/SM	~19,000	~20
L2 cache	40 MB	~4,000	~150
HBM (global)	80 GB	~1,500	~400

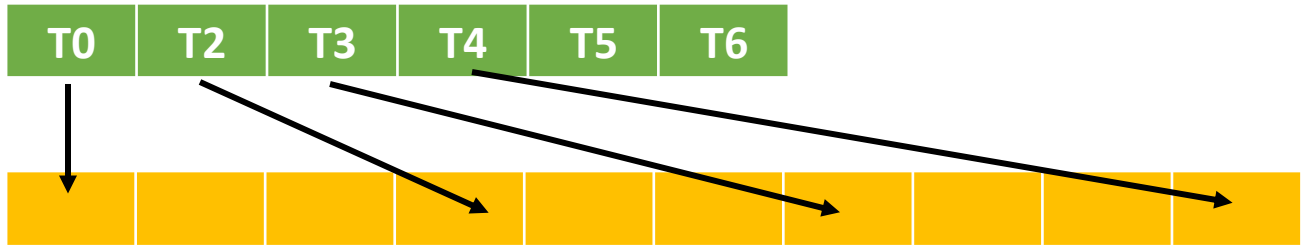
Approximate data for A100 from NVIDIA GTC “How GPU Computing Works”



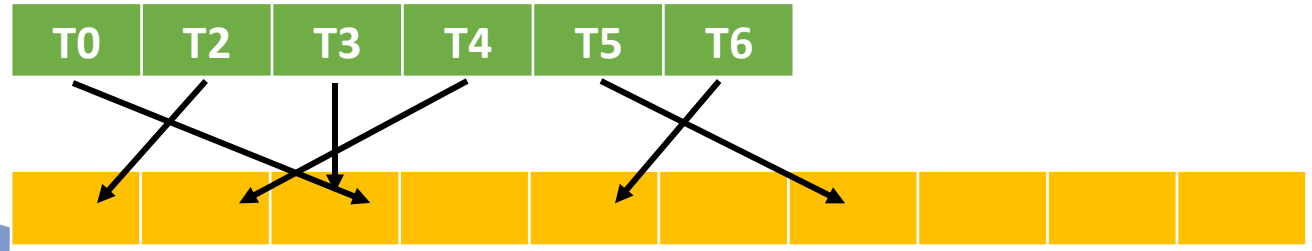
Global memory access patterns



Coalesced memory access pattern. Consecutive threads load consecutive memory locations. Loads can be done in one transaction – most efficient.



Strided memory access pattern. Constant offset, here stride size is 2. Performance decreases with stride size.



Random memory access pattern. Worst for performance.

Memory access lab

ssh **your_user_name**@vista.tacc.utexas.edu

idev –p gh

(full syntax and file location)

idev -p gh -r PerfCounter_testing –A TRA24006

cds && cp ~train00/IntroToCUDA.tar .

tar -xvf IntroToCUDA

cd IntroToCUDA/2_MemoryAccess

nvcc –o stride stride.cu

./stride

Strided memory access kernel

```
__global__ void stridedMemAccess(float *d_in, int stride){
        int tid = (blockDim.x * blockIdx.x + threadIdx.x) * stride;
        d_in[tid] = d_in[tid] + stride;
}

for (int stride = 1; stride <= 32; stride++) {
        stridedMemAccess<<<gridSize, blockSize>>>(d_in,stride);
}
```

	tid = 0	tid = 1	tid = 2	tid = 3	tid = 4	tid = 5
stride=1	0	1	2	3	4	5
stride=2	0	2	4	6	8	10
stride=3	0	5	10	15	20	25

```
Processing 1310720 fp32 elements
Stride: 1, Bandwidth (GB/s) 1020.809984
Stride: 2, Bandwidth (GB/s) 1067.361536
Stride: 3, Bandwidth (GB/s) 880.860224
Stride: 4, Bandwidth (GB/s) 753.287360
Stride: 5, Bandwidth (GB/s) 672.854208
Stride: 6, Bandwidth (GB/s) 580.992896
Stride: 7, Bandwidth (GB/s) 513.604992
Stride: 8, Bandwidth (GB/s) 441.617248
Stride: 9, Bandwidth (GB/s) 400.097664
Stride: 10, Bandwidth (GB/s) 377.946944
Stride: 11, Bandwidth (GB/s) 351.588000
Stride: 12, Bandwidth (GB/s) 334.026496
Stride: 13, Bandwidth (GB/s) 314.774240
Stride: 14, Bandwidth (GB/s) 289.469952
Stride: 15, Bandwidth (GB/s) 275.824928
Stride: 16, Bandwidth (GB/s) 260.270048
Stride: 17, Bandwidth (GB/s) 251.867792
Stride: 18, Bandwidth (GB/s) 242.366848
Stride: 19, Bandwidth (GB/s) 231.739744
Stride: 20, Bandwidth (GB/s) 226.768176
Stride: 21, Bandwidth (GB/s) 215.154304
Stride: 22, Bandwidth (GB/s) 206.088048
Stride: 23, Bandwidth (GB/s) 192.526432
Stride: 24, Bandwidth (GB/s) 183.368768
Stride: 25, Bandwidth (GB/s) 174.855920
Stride: 26, Bandwidth (GB/s) 169.343680
Stride: 27, Bandwidth (GB/s) 161.259840
Stride: 28, Bandwidth (GB/s) 155.372208
Stride: 29, Bandwidth (GB/s) 150.796128
Stride: 30, Bandwidth (GB/s) 148.810176
Stride: 31, Bandwidth (GB/s) 144.416032
Stride: 32, Bandwidth (GB/s) 143.467600
```

CUDA streams – even more parallelism

Kernel launches are **asynchronous** with respect to the host (CPU). CPU starts the kernel and then moves to the next instruction.

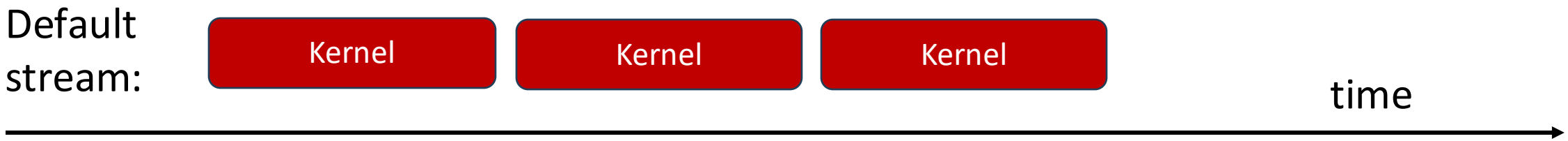
Regular data copies (cudaMemcpy) are **synchronous** or blocking with respect to the host – CPU waits for the data transfer to complete before moving to the next instruction. Asynchrounous data transfer requires pinned memory and using cudaMemcpyAsync.

CUDA streams

Streams are instructions pipelines for the GPU.

Instructions within the same stream will execute sequentially, in order they were issued.
Instructions in different streams can run in parallel if sufficient resources on the GPU are available.

Sequential code



Code with multiple steams

Creating streams and memory copies

```
cudaStream_t stream1;

cudaStreamCreat(&stream1);

kernel_name<<<gridSize,blockSize,0,stream1>>>(…);

cudaStreamSynchronize(&stream1);

cudaStreamDestroy(&stream1);


cudaMallocHost((void**)&a_pinned, sizeof(int) * N);

cudaMemcpyAsync(dest, source, size, direction, stream1);
```

ssh **your_user_name**@vista.tacc.utexas.edu

idev –p gh

(full syntax and file location)

idev -p gh -r PerfCounter_testing –A TRA24006

cds && cp ~train00/IntroToCUDA.tar .

tar -xvf IntroToCUDA

cd IntroToCUDA/3_Streams

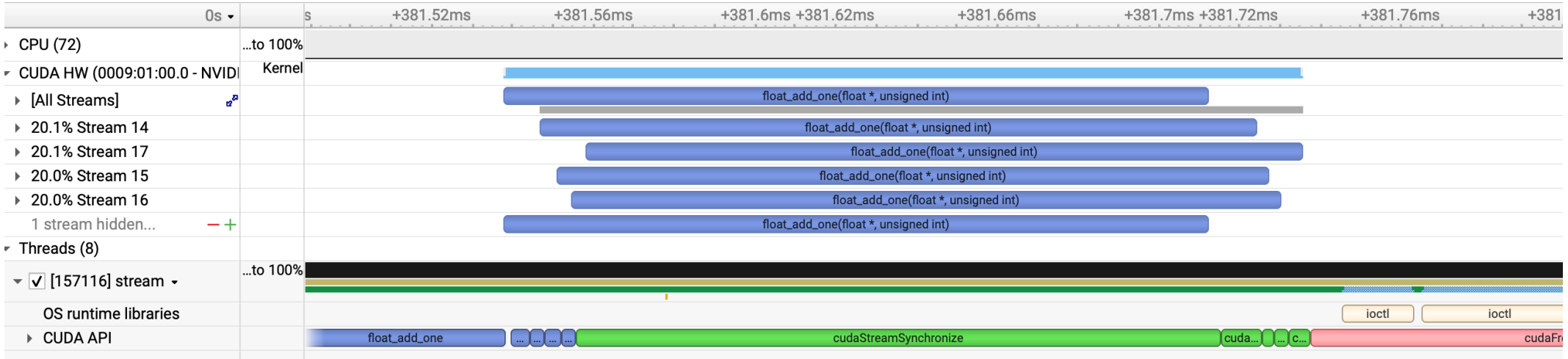
nvcc –o streams streams.cu

./streams [no. of streams] [blocks per grid]

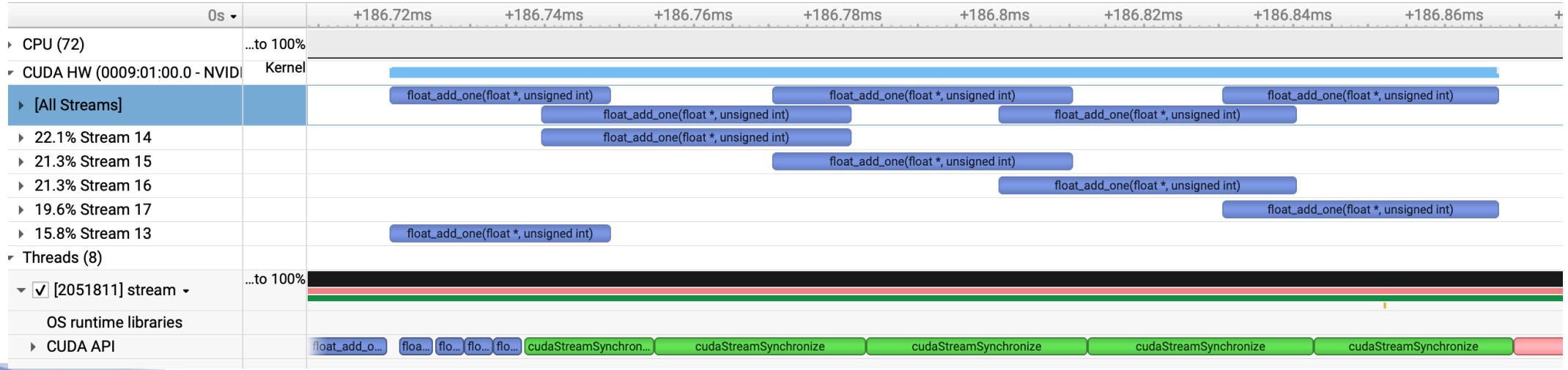
nsys profile –o streamProf

Code reproduced with modifications from:
https://leimao.github.io/blog/CUDA-Kernel-Execution-Overlap/

<<< 32, 1024 >>>



<<< 1024, 1024 >>>



Shared memory

On-chip memory, x100 faster than global memory

Manually managed part of L1 cache

Visible to all threads within the same block

When to use shared memory:

- Communication within the block
- Combining multiple writes to global memory into a single transaction
- Data re-use within the block

Can cause race condition, need for explicit synchronization barrier within the block with *__syncthreads()*

Using shared memory

Static shared memory - known at compile time – declared within the kernel

```
__global__ static_shered_mem (…){

        __shared__ int shared_array[512]; // dynamic shared memory
… }

static_shered_mem<<<BLOCK_SIZE, GRID_SIZE>>> (…);
```

Dynamic shared memory - known at runtime – size passed to the kernel

```
__global__ dynamic_shered_mem(args){

        extern __shared__ int shared_array[]; // dynamic shared memory
… }

dynamic_shered_mem<<<BLOCK_SIZE, GRID_SIZE, SHARED_MEM_SIZE>>> (…);
```

Shared memory example

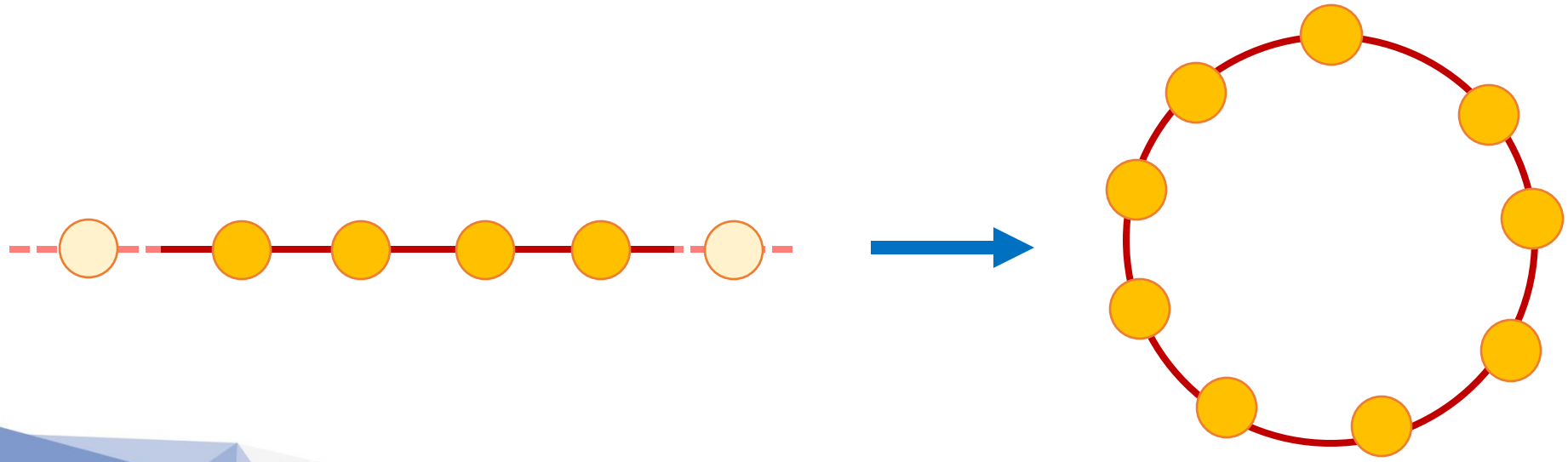
Repulsive (spring potential) particles in a 1D periodic box → circle

Interactions with *N* nearest neighbors

No. boxes = no. blocks | No. particles in the box = no. threads per block

Kernel with and without shared memory + CPU version

Spacer variable changes stride between data entries



ssh **your_user_name**@vista.tacc.utexas.edu

idev –p gh

(full syntax and file location)

idev -p gh -r PerfCounter_testing –A TRA24006

cds && cp ~/train00/IntroToCUDA.tar .

tar -xvf IntroToCUDA

cd IntroToCuda/4_SharedMemory

nvcc –o forceGPU forceGPU.cu

./forceGPU

GPU kernel – no shared memory

```
__global__ void ForceGPU(float *d_force, float *d_pos, int N, float A, float B, float B2, int spacer)
{

   int tid = threadIdx.x;
   int offset = blockDim.x * blockIdx.x;
   d_force[tid + offset] = 0.0;
   float my_pos = d_pos[spacer*(tid + offset)];

   float f = (abs(d_pos[spacer*((tid+1)%N + offset)] - my_pos) - B) -     (abs(d_pos[spacer*((N + tid-1)%N + offset)] -
my_pos) - B)+ ...

   d_force[tid + offset] = A/B2 * f;
}
```

Shared memory kernel

```
__global__ void ForceGPUShared(float *d_force, float *d_pos, int N, float A, float B, float B2, int spacer)
{
  extern __shared__ float shared_pos[];

    int tid = threadIdx.x;
    int gid = blockDim.x * blockIdx.x + threadIdx.x;

    shared_pos[tid] = d_pos[gid * spacer];
    d_force[gid] = 0.0;
    float my_pos = d_pos[gid * spacer];

    __syncthreads();

    d_force[gid] = A/B2 * ((abs(shared_pos[(tid+1)%N] - my_pos) - B) - (abs(shared_pos[(N + tid-1)%N] - my_pos) - B) +
...}
```

Calling the kernel with shared memory

```
int gridSize = 100;
int blockSize = 1024;
int N = blockSize * gridSize;

ForceGPUShared<<<gridSize, blockSize, blockSize*sizeof(float)>>>(args);
```

CPU timing

```
#include <chrono>

auto start_cpu = std::chrono::high_resolution_clock::now();

ForceCPU(args);

auto end_cpu = std::chrono::high_resolution_clock::now();

cpu_time= std::chrono::duration_cast<std::chrono::nanoseconds>(end_cpu - start_cpu).count();
```

200 blocks | 1024 threads | spacer = 32

=== Performance Statistics ===
CPU time: 3.282658 ms
GPU time: 0.012021 ms
GPU time with shared memory: 0.007261 ms

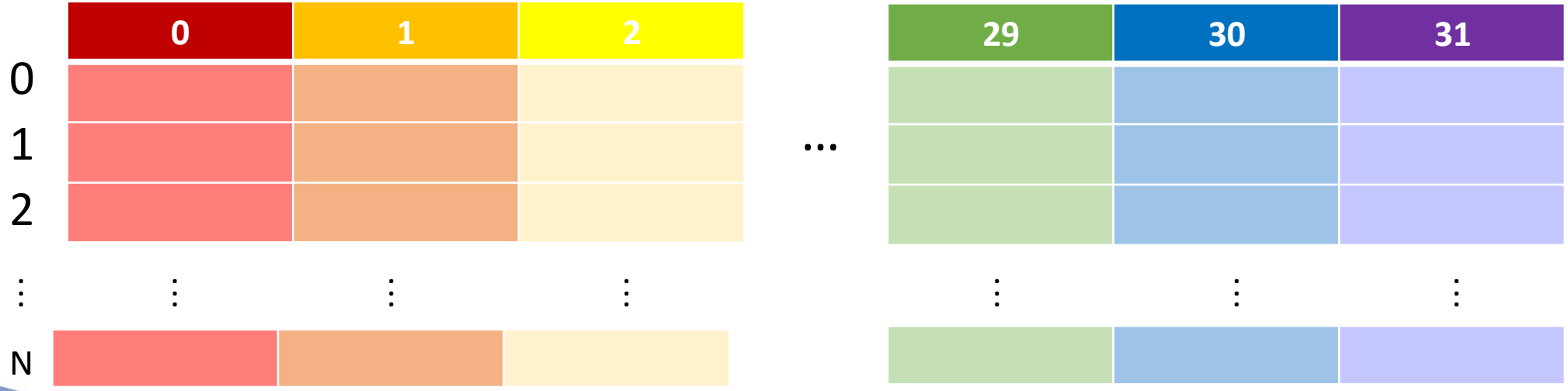
200 blocks | 1024 threads | spacer = 1
=== Performance Statistics ===
CPU time: 3.263707 ms
GPU time: 0.005230 ms
GPU time with shared memory: 0.005226 ms

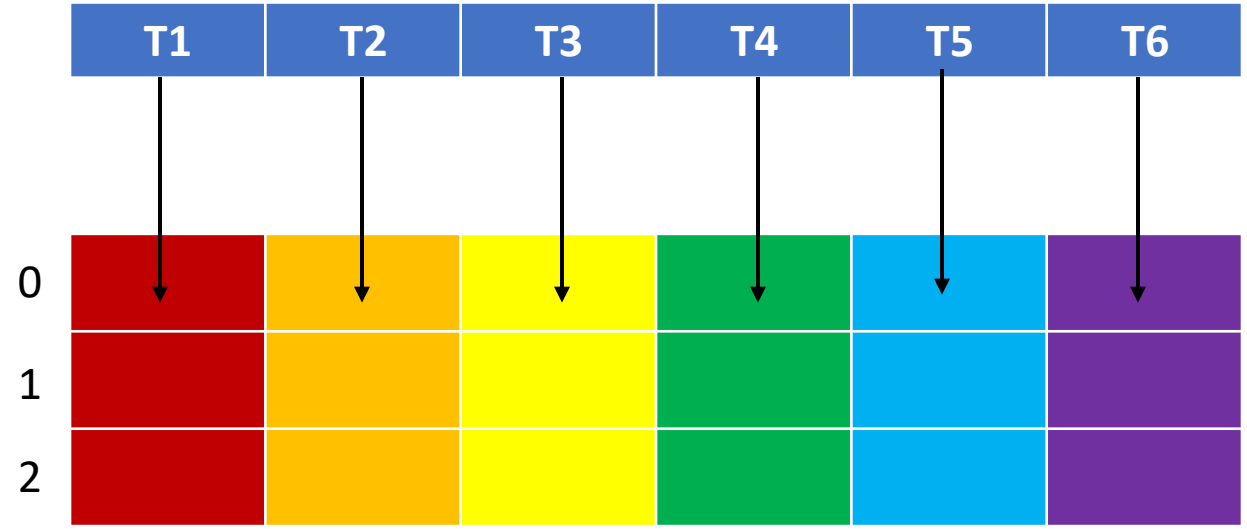
Shared memory and bank conflicts

Shared memory arranged in 32 banks, 4 bytes each.

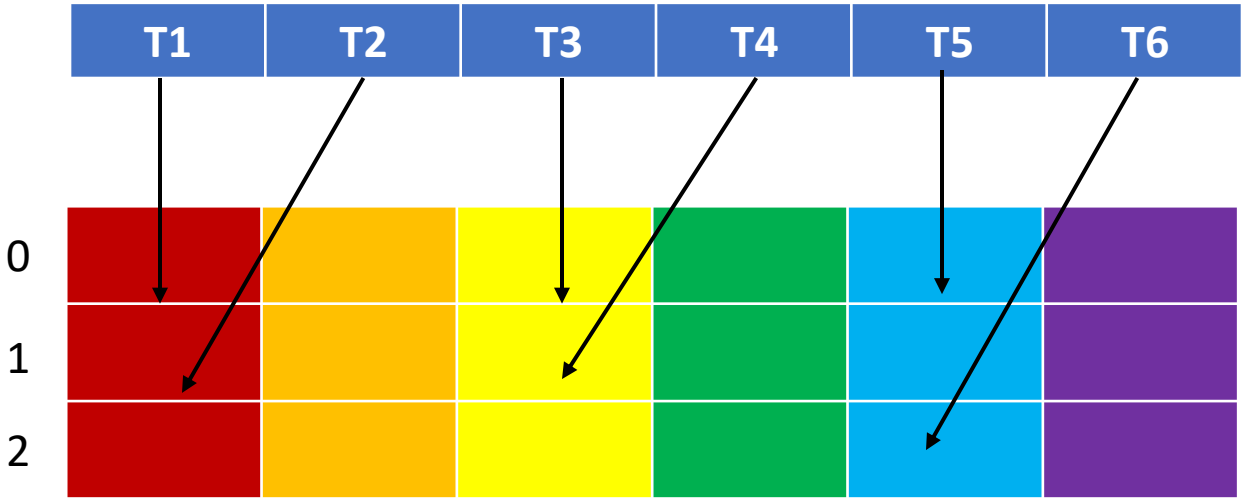
One row (128 bytes) are accessed per memory transaction

Only one row can be accessed at once. If two threads request data in the same bank, we have a bank conflict, two operations needed.





No bank conflicts, all data can be loaded in one transaction



Separate transactions needed for all rows

Exercise

- Compare the version with and without the shared memory.
- How changing the stride (*spacer variable*) affects the performance of both GPU kernels?
- Vary the number of blocks and block size. How changing these parameters affects the GPU and CPU time?