EURO
LUXEMBOURG

Introduction to GPU programming

Dr. Ezhilmathi Krishnasamy
Parallel Computing and Optimization Group (PCOG), University of Luxembourg (UL), Luxembourg

EURO
LUXEMBOURG

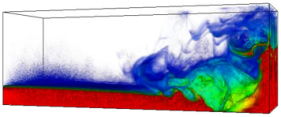
What we are going to discuss

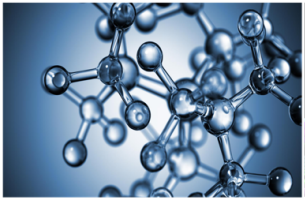
- GPU and CPU architecture overview and comparison
  - streaming multiprocessors, memory hierarchy, threads blocks, etc,.
- CUDA programming model
  - programming structure, thread hierarchy, device call, etc,.
- Memory management
  - unified memory, explicit memory copy, etc,.
- Examples in numerical linear algebra
  - vector multiplication, vector addition, etc,.
- A quick demo session with some examples

Motivation: Why we need supercomputers

- Powerful computers will help to unlock the secrets in science and engineering

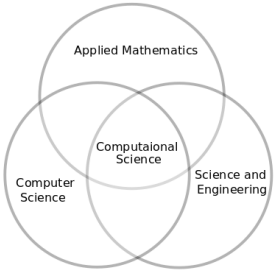
- Astrophysics
- CFD:turbulence
- Bioinformatics
- Material science

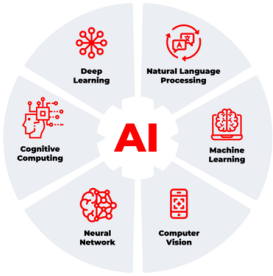




Motivation: Why we need supercomputers

- We need to do lots of arithmetic computation in science & engineering and artificial intelligence
- For example, in science and engineering, problems are defined by partial differential equations (PDEs)
- PDEs are converted into a system of equations by using numerical methods (e.g., finite difference and finite element methods), where we need to find the values for the unknown variables
- Similarly, in artificial intelligence, we end up solving matrices and vectors

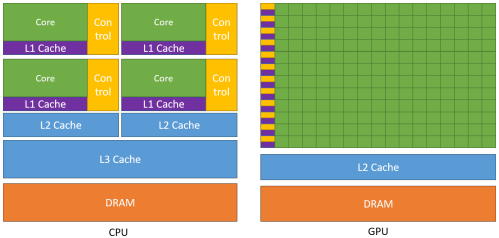




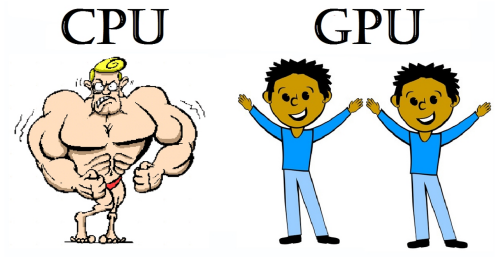
$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Important differences between CPU and GPU

- GPU has many cores compared to CPU
- But on the other hand, the CPU's frequency is higher than the GPU. That makes the CPU faster in computing compared to GPU
  - Intel® Core™ i7-10700K Processor base frequency is 3.80 GHz, whereas, Nvidia Ampere has 0.765 GHz
- However, GPU can handle many threads in parallel, which can process many data in parallel
- In the GPU, cores are grouped into GPU Processing Clusters (GPCs), and each GPCs has its own Streaming Multiprocessors (SMs) and Texture Processor Clusters (TPCs)
- Nvidia (microarchitecture): Tesla (2006), Fermi (2010), Kepler (2012), Maxwell (2014), Pascal (2016), Volta (2017), Turing (2018), and Ampere (2020)
- Video Link: Mythbusters Demo GPU versus CPU



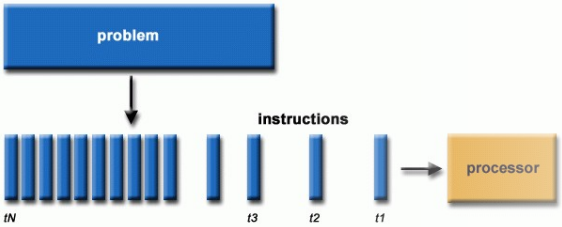
Source:Nvidia: CUDA programming



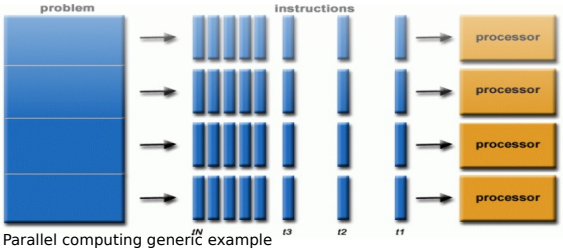
Serial programming vs. parallel programming

- Serial programming:
  - An entire problem can be divided in to discrete series of instructions
  - All the instructions are executed one by one
  - Executed by single thread or processor
  - Only one instruction can be executed at the same time
- Parallel programming
  - An entire problem can be divided into discrete parts such way that it can be solved concurrently
  - Each part may have set of instructions
  - Each parts instructions are executed on different thread/processor
  - Since it is a parallel execution, a target problem needs to be controlled/coordinated
- CPU, GPU, and other parallel processor can perform the parallel computing

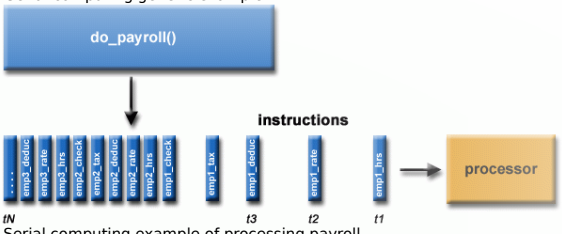
Serial programming vs. parallel programming



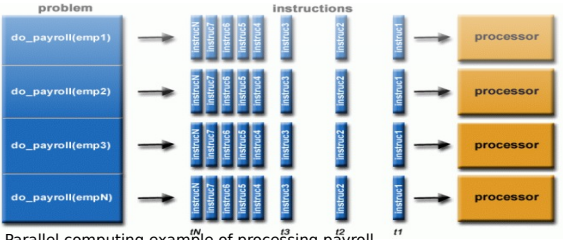
Serial computing generic example



Parallel computing generic example



Serial computing example of processing payroll



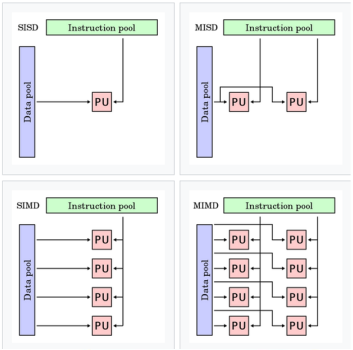
Parallel computing example of processing payroll

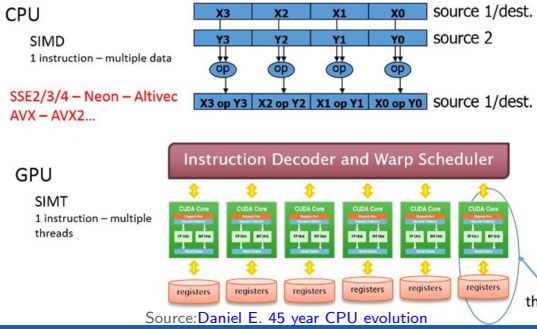
Source: HPC LLNL

GPU architecture

- Computer architecture is characterized by 4 according to Flynn's taxonomy
  - Single instruction stream, single data stream (SISD)
  - Single instruction stream, multiple data streams (SIMD)
  - Multiple instruction streams, single data stream (MISD)
  - Multiple instruction streams, multiple data streams (MIMD)

- GPUs are based on Single Instruction Multiple Threads (SIMT)

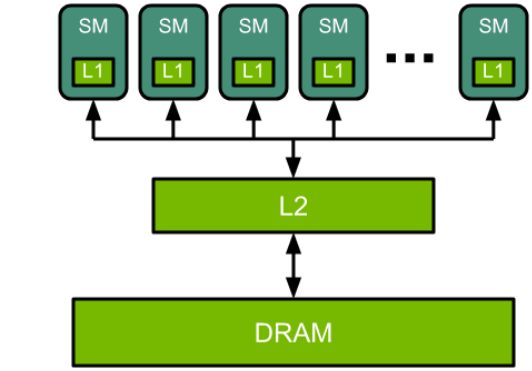




Source:Daniel E. 45 year CPU evolution

GPU architecture

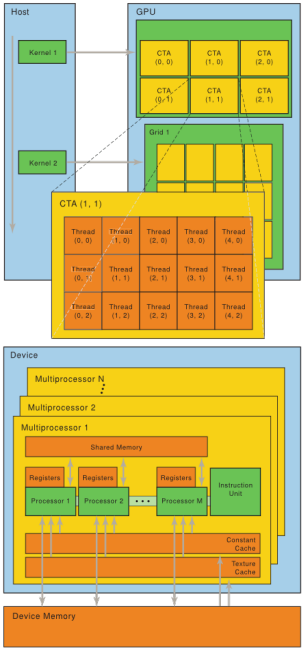
- Ampere GPU had seven GPCs, 42 TPCs, and 84 SMs.
- Volta GPU has six GPCs, each GPC has a seven TPCs (each including two SMs), and 14 SMs.
- Each SMs has L1 cache (up to 128 KB) and L2 (up to 6144 KB) cache is shared between the GPCs.
- RT (Ray Tracing) cores dedicated to do the ray-tracing rendering math computation.
- Tensor Cores: provides the speedups for AI neural network training computation.
- Programmable Shading Cores, which has a CUDA cores.



Source: Nvidia: deep learning

GPU architecture

- SIMT enables programmers to achieve thread-level parallelism in streaming multiprocessors (SMs)
- The multiprocessor *occupancy* is the ratio of active warps to the maximum number of warps supported on the GPU's multiprocessor
- SMs in the GPU are based on the scalable array multi-thread, which allows grid and thread blocks of 1D, 2D, and 3D data
- Programmers can write the grid and block size to create a thread when executing the device kernel; this thread block is typically called a cooperative thread array (CTA)
- A parallel execution is happening in the SMs via *warps* and one warp contains 32 threads



Source: Nvidia: Parallel Thread Execution

EURO
LUXEMBOURG

Usage of compute capabilities in different Nvidia GPU architecture

Compute capability (flag)	**Architecture support**
sm_35, and sm_37	Basic features + Kepler support + Unified memory programming + Dynamic parallelism support
sm_50, sm_52 and sm_53	+ Maxwell support
sm_60, sm_61, and sm_62	+ Pascal support
sm_70 and sm_72	+ Volta support
sm_75	+ Turing support
sm_80, sm_86 and sm_87	+ NVIDIA Ampere GPU architecture support

Thread organization

- Threads are organized within a Grids and Blocks. These Grids and Blocks can be in 1D, 2D or 3D. And these are declared as *dim3*
- Example: 2D grid and thread block

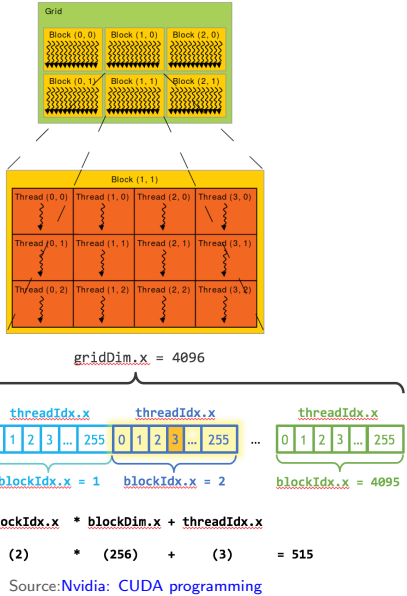
```
dim3 Grid(3, 2, 1); # two dimentional grid
dim3 Block(4, 3, 1); # two dimentinal thread block
```

- Example: 1D grid and thread block

```
dim3 Grid(4096, 1, 1); # one dimentional grid
dim3 Block(256, 1, 1); # one dimentinal thread block
```

- Example: calling thread block in the main program

```
Hello_world(); # calling c function
Hello_world<<<Grid, Block>>>(); calling cuda device function
```



Source:Nvidia: CUDA programming

EURO LUXEMBOURG

Thread organization

- Dimension variables:
  - **gridDim** specifies the number of blocks in the grid
  - **blockDim** specifies the number of threads in each block
- Index variables:
  - **blockIdx** gives the index of the block in the grid
  - **threadIdx** gives the index of the thread within the block

CUDA API

- *cudaMalloc()* allocates device memory
- *cudaMemcpy()* transfers data to or from a device
- *cudaFree()* frees device memory that is no longer in use
- *__syncthreads()* synchronizes threads within a block
- *cudaDeviceSynchronize()* effectively synchronizes all threads in a grid
- *cudaMallocManaged()* for allocating unified memory

Major comparison between Turing vs. Ampere

Graphics Card	GeForce RTX 2080 Founders Edition	GeForce RTX 3080 10GB Founders Edition
GPU Codename	TU104	GA102
GPU Architecture	**Nvidia Turing**	**Nvidia Ampere**
GPCs	6	6
TPCs	23	34
SMs	46	68
CUDA Cores / SM	64	128
CUDA Cores / GPU	2944	8704
Tensor Cores / SM	8 (2nd Gen)	4 (3rd Gen)
Tensor Cores / GPU	368	272 (3rd Gen)
RT cores	46 (1st Gen)	68 (2nd Gen)

Source:Nvidia Ampere

Compute capabilities for latest Nvidia GPUs

Data Center GPU	**Nvidia V100**	**Nvidia A100**	**Nvidia H100**
GPU architecture	Nvidia Volta	Nvidia Ampere	Nvidia Hopper
Compute Capability	7	8	9
Thread / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks (CTAs) / SM	32	32	32
Max Threads Blocks / Thread Block Clusters	NA	NA	NA
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Thread Block	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size (#of threads)	1024	1024	1024
FP32 Cores / SM	64	64	64
Ratio of SM Registers to FP32 Cores	1024	1024	1024
Shared Memory Size / SM	Configurable up to 96 KB	Configurable up to 164 KB	Configurable up to 228 KB

Source:Nvidia H100

CUDA function qualifiers and variable memory space specifiers

Qualifier	Description
__device__	These functions are executed only from the device and callable only from device
__global__	These functions are executed from the device, and it can be callable from the host and device (only for compute capabilities 3.2 or higher)
__host__	These functions are executed from a host, and callable only from the host
__noninline__ *__forceinline__*	Compiler directives instruct the functions to be inline or not inline

Variable	Memory	Scope	Lifetime
__device__	Global	Grid (entire grid of thread blocks)	Application
__constant__	Constant	Grid (entire grid of thread blocks)	Application
__shared__	Shared	Block (within a thread block)	Block

Hello world

- Run a part or entire application on the GPU
- Call *cuda_function* on device
- It should be called using function qualifier *__global__*
- Calling the device function on the main program:
  - C/C++ example, *c_function()*
  - CUDA example, *cuda_function<<<1,1>>>()* (just using 1 thread)
- <<< >>>, specify the threads blocks within the bracket
- Make sure to synchronize the threads
  - __syncthreads(); synchronizes all the threads within a thread block
  - CudaDeviceSynchronize(); synchronizes a kernel call in host
- Most of the CUDA API are synchronized call by default (but sometimes it is good to call explicit synchronized call to avoid error in the computation)

```c
// hello-world.c
#include <stdio.h>
void c_function()
{
  printf("Hello World!\n");
}

int main()
{
  c_function();
  return 0;
}
```

```c
// hello-world.cu
#include <stdio.h>
__global__ void cuda_function()
{
   printf("Hello World from GPU!\n");
    __syncthreads();                // to synchronize all threads
}

int main()
{
    cuda_function<<<1,1>>>();
    cudaDeviceSynchronize();      // to synchronize device call
    return 0;
}
```

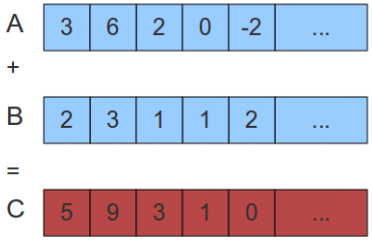
Vector addition

- Memory allocation on both CPU and GPU

```
// Initialize the memory on the host
float *a, *b, *out;

// Allocate host memory
a   = (float*)malloc(sizeof(float) * N);
b   = (float*)malloc(sizeof(float) * N);
out = (float*)malloc(sizeof(float) * N);

// Initialize the memory on the device
float *d_a, *d_b, *d_out;

// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);
```

- Fill values for host vectors a and b

```
// Initialize host arrays
for(int i = 0; i < N; i++)
  {
    a[i] = 1.0f;
    b[i] = 2.0f;
  }
```



Vector addition

- Transfer initialized value from CPU to GPU

```
// Transfer data from host to device memory
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
```

- Creating a 2D thread block

```
// Thread organization
dim3 dimGrid(1, 1, 1);
dim3 dimBlock(8, 8, 1);
```

- Calling the kernel function

```
// execute the CUDA kernel function
  vector_add<<<dimGrid, dimBlock>>>(d_a, d_b, d_out, N);
```

- Copy back computed value from GPU to CPU

```
// Transfer data back to host memory
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
```

Vector addition

- Vector addition function call

```
// GPU function that adds two vectors
__global__ void vector_add(float *a, float *b,
           float *out, int n)
{

  int i = blockIdx.x * blockDim.x * blockDim.y +
    threadIdx.y * blockDim.x + threadIdx.x;
  // Allow the   threads only within the size of N
  if(i < n)
    {
      out[i] = a[i] + b[i];
    }

  // Synchronice all the threads
  __syncthreads();
}
```

- Release the host and device memory

```
// Deallocate device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_out);

// Deallocate host memory
free(a);
free(b);
free(out);
```

- Source:Vector-Addition.cu

Matrix multiplication

Matrix multiplication function in C/C++

```
float * matrix_mul(float *h_a, float *h_b, float *h_c, int width)
{
  for(int row = 0; row < width ; ++row)
    {
      for(int col = 0; col < width ; ++col)
        {
          float single_entry = 0;
          for(int i = 0; i < width ; ++i)
            {
              single_entry += h_a[row*width+i] * h_b[i*width+col];
            }
          h_c[row*width+col] = single_entry;
        }
    }
  return h_c;
}
```

Source: Matrix-Multiplication.cu

Matrix multiplication function in CUDA

```
__global__ void matrix_mul(float* d_a, float* d_b,
                           float* d_c, int width)
{

  int row = blockIdx.x * blockDim.x + threadIdx.x;
  int col = blockIdx.y * blockDim.y + threadIdx.y;

  if ((row < width) && (col < width))
    {
      float single_entry = 0;
      // each thread computes one
      // element of the block sub-matrix
      for (int i = 0; i < width; ++i)
        {
          single_entry += d_a[row*width+i]*d_b[i*width+col];
        }
      d_c[row*width+col] = single_entry;
    }
}
```

Matrix multiplication

- Allocating the CPU and GPU memory for A,B, and C matrix

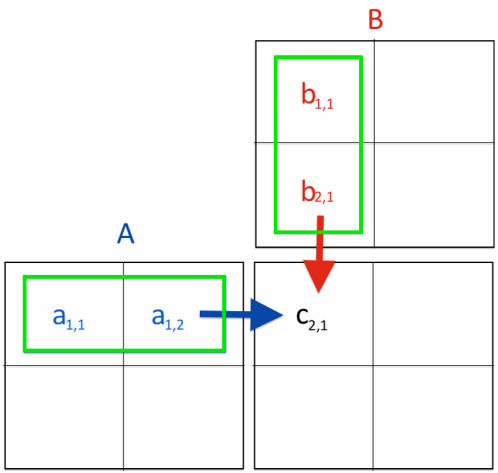
```
// Initialize the memory on the host
float *a, *b, *c;

// Allocate host memory
a = (float*)malloc(sizeof(float) * (N*N));
b = (float*)malloc(sizeof(float) * (N*N));
c = (float*)malloc(sizeof(float) * (N*N));

// Initialize the memory on the device
float *d_a, *d_b, *d_c;

// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * (N*N));
cudaMalloc((void**)&d_b, sizeof(float) * (N*N));
cudaMalloc((void**)&d_c, sizeof(float) * (N*N));
```

- Transfer initialized A and B matrix from CPU to GPU

```
// Transfer data from host to device memory
cudaMemcpy(d_a, a, sizeof(float) * (N*N), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * (N*N), cudaMemcpyHostToDevice);
```

- 2D thread block for indexing x and y

```
// Thread organization
int blockSize = 32;
dim3 dimBlock(blockSize,blockSize,1);
dim3 dimGrid(ceil(N/float(blockSize)),ceil(N/float(blockSize)),1);
```



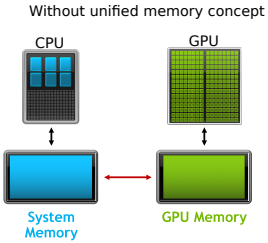
Unified Memory

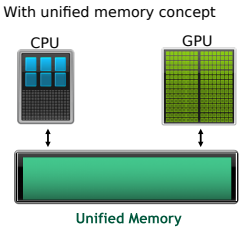
Without unified memory

- Allocate the host memory
- Allocate the device memory
- Initialize the host value
- Transfer the host value to device memory location
- Do the computation using the CUDA kernel
- Transfer the data from the device to host
- Free device memory
- Free host memory

With unified memory

- ~~Allocate~~ ~~the host memory~~
- Allocate the device memory
- Initialize the host value
- ~~Transfer the host value to device memory location~~
- Do the computation using the CUDA kernel
- ~~Transfer the data from the device to host~~
- Free device memory
- ~~Free host memory~~





Source: Vector-Addition-Unified.cu

Unified Memory

Use *cudaMallocManaged()*

```
/*
// Initialize the memory on the host
float *a, *b, *out;

// Allocate host memory
a   = (float*)malloc(sizeof(float) * N);
b   = (float*)malloc(sizeof(float) * N);
out = (float*)malloc(sizeof(float) * N);
*/

// Initialize the memory on the device
float *d_a, *d_b, *d_out;

// Allocate device memory
cudaMallocManaged(&d_a, sizeof(float) * N);
cudaMallocManaged(&d_b, sizeof(float) * N);
cudaMallocManaged(&d_out, sizeof(float) * N);
```

Do not forget to call *cudaDeviceSynchronize()* after a kernel call

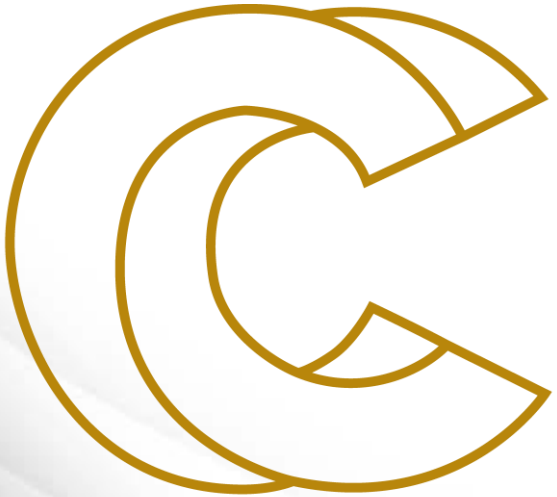
EURO
LUXEMBOURG

Thank you

If you any question or research collaboration, please contact
ezhilmathi.krishnasamy@uni.lu

If you interested to learn more about CUDA and OpenACC programming, please refer to PRACE MOOC: GPU Programming for Scientific Computing and Beyond (given by Prof. Pacal Bouvry and Dr. Ezhilmathi Krishnasamy)

Workshop - Programming on Accelerators



EURO

Dr. Özcan DÜLGER
Computer Engineering, Middle East Technical University
Computer Engineering, Artvin Coruh University

23 May 2022

ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

Performance Optimization and Efficiency

Ozcan Dulger, NCC Turkey

EURO

Contents:

- Memory Coalesced Access to Global Memory
- Device Occupancy and SM Efficiency
- Warp Divergence

Ozcan Dulger, NCC Turkey

EURO

Tesla K40 Board

Property	Value	Property	Value
Architecture	Kepler	Global Memory	11520 MB
Number of SMX	15	Shared Memory	49152 Byte
CUDA Core	2880	L2 Cache	1572864 Byte
Core Clock	745 MHz	Segment Size	128 Byte
Max. Thread / SMX	2048	Warp Size	32
Max. Thread / Block	1024	Max. Block / SMX	16

Performance Optimization and Efficiency

Ozcan Dulger, NCC Turkey

EURO

Memory Coalesced Access:

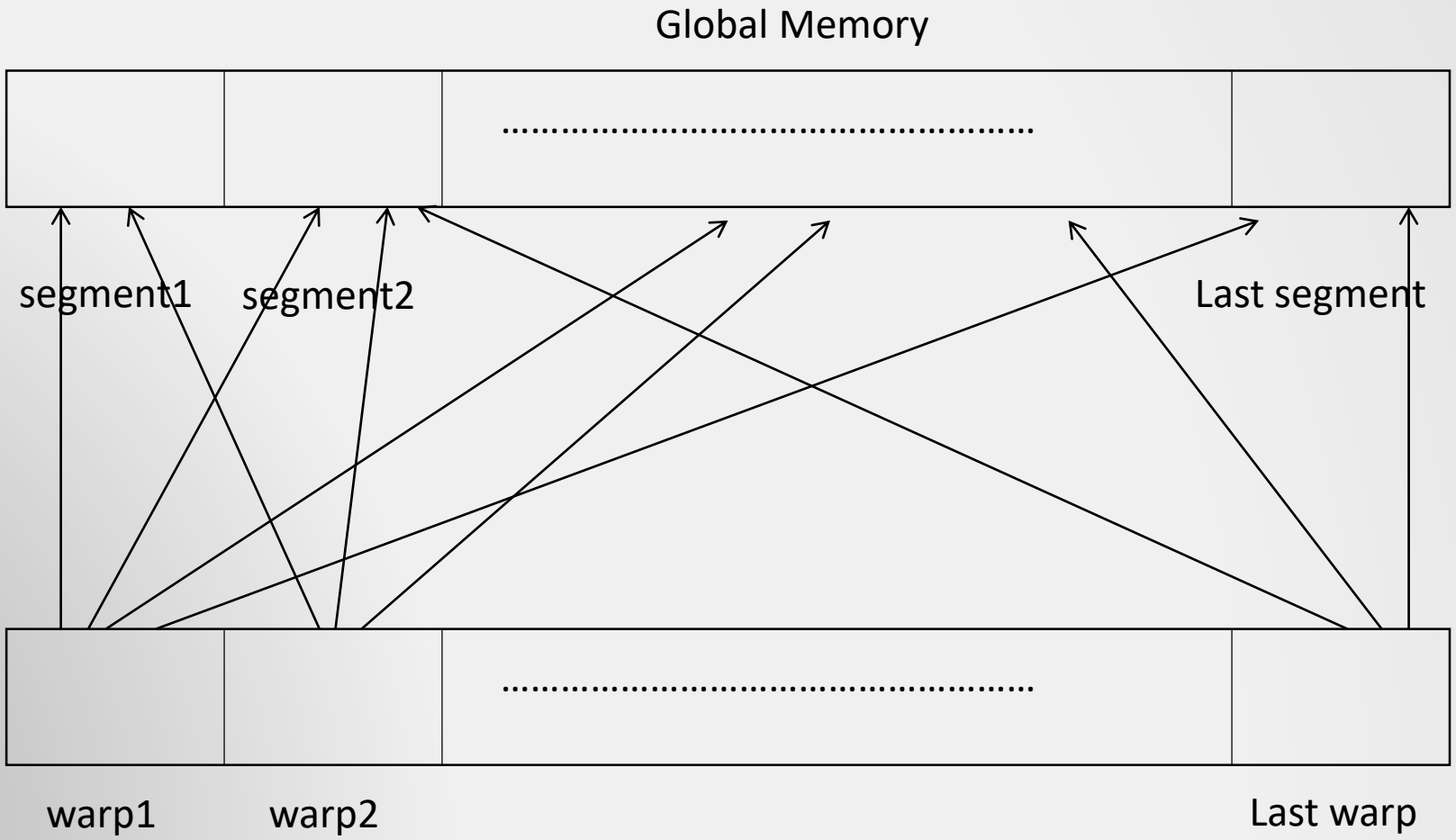
- Reading from or writing to global memory performs segment by segment
- The threads in a warp are physically related to each other. That means a warp completes its instruction when all the threads in the warp complete the instruction
- In global memory operations, if the threads in the warp access to the different segments of the global memory, the operations become serial

Performance Optimization and Efficiency

Ozcan Dulger, NCC Turkey

EURO

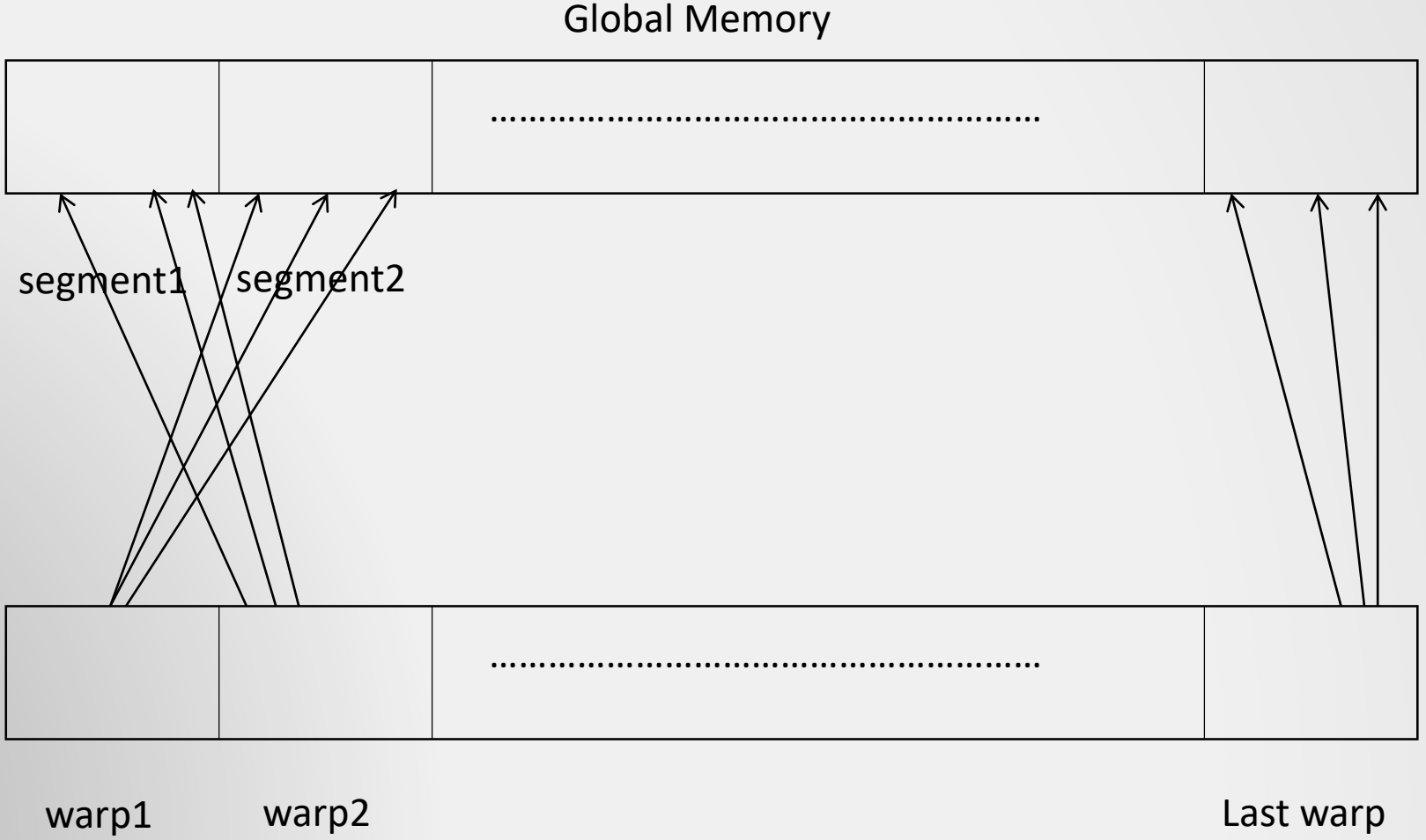
Non-Coalesced Access



Performance Optimization and Efficiency

Ozcan Dulger, NCC Turkey

Memory Coalesced Access



```cpp
__global__ void full_coalesced_access(float *A,float *B,float *C)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    for(int i=0;i<100;i++)
    {
        C[tid] = A[tid] + B[tid];//Vector addition
    }
}

__global__ void non_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs,unsigned int NP)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state;// State of the generator
    RNGs.load(state,tid);// Loading the state
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = curand(&state)/NP;// Generate a random number between 0 and size-1
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays

    for(int counter = 0;counter < data_size; counter++)
    {
        A_host[counter] = counter+1;//Assigning numbers from 1 to size
        B_host[counter] = counter+2;//Assigning numbers from 2 to size+1
    }

    cudaMemcpy(A_GPU,A_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
    cudaMemcpy(B_GPU,B_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);

    unsigned int NTB = 1024;//Number of threads in a block
    unsigned int NP_data_size = (unsigned long int)pow(2,32)/data_size;// Number of partitions in a period for 'data_size'

    dim3 threadsPerBlock(NTB);//Number of threads in a block
    dim3 numBlocks(data_size/NTB);//Number of blocks in a grid

    curandInitializer RNGs(data_size);//Creating a generator for 'non_coalesced_access' kernel

    full_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'full_coalesced_access' kernel
    non_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,RNGs,NP_data_size);//Launching 'non_coalesced_access' kerne
    cudaDeviceSynchronize();//Waits until vector_add kernel completes its run
}
```

Load in one transaction

Load the state of the generator of each thread from global memory as a coalesced way

Load in at most 32 transactions

XORWOW Generators

Example of Memory Coalesced Access

Note: We use cudaEventRecord in order to measure the kernel execution times

EURO

```cpp
__global__ void full_coalesced_access(float *A,float *B,float *C)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    for(int i=0;i<100;i++)
    {
        C[tid] = A[tid] + B[tid];//Vector addition
    }
}

__global__ void non_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs,unsigned int NP)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state;// State of the generator
    RNGs.load(state,tid);// Loading the state
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = curand(&state)/NP;// Generate a random number between 0 and size-1
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 32768;//Data size
    float *A_host,*B_host,*C_host;//Host Arrays
```

Metrics:

gld_transactions: Number of global memory load transactions

gld_transactions_per_request: Average number of global memory load transactions performed for each global memory load

```
Exec. Time of 'full_coalesced_access' kernel = 0.000113152
Exec. Time of 'non_coalesced_access' kernel = 0.00168758
Speed Up = 14.9143X
```

```
==1430== Profiling result:
==1430== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: full_coalesced_access(float*, float*, float*)
          1                        gld_transactions                  Global Load Transactions      204800      204800      204800
          1              gld_transactions_per_request       Global Load Transactions Per Request    1.000000    1.000000    1.000000
    Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
          1                        gld_transactions                  Global Load Transactions     6461956     6461956     6461956
          1              gld_transactions_per_request       Global Load Transactions Per Request   30.633514   30.633514   30.633514
```

```cpp
__global__ void full_coalesced_access(float *A,float *B,float *C)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    for(int i=0;i<100;i++)
    {
        C[tid] = A[tid] + B[tid];//Vector addition
    }
}

__global__ void non_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs,unsigned int NP)
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state;// State of the generator
    RNGs.load(state,tid);// Loading the state
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = curand(&state)/NP;// Generate a random number between 0 and size-1
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays
```

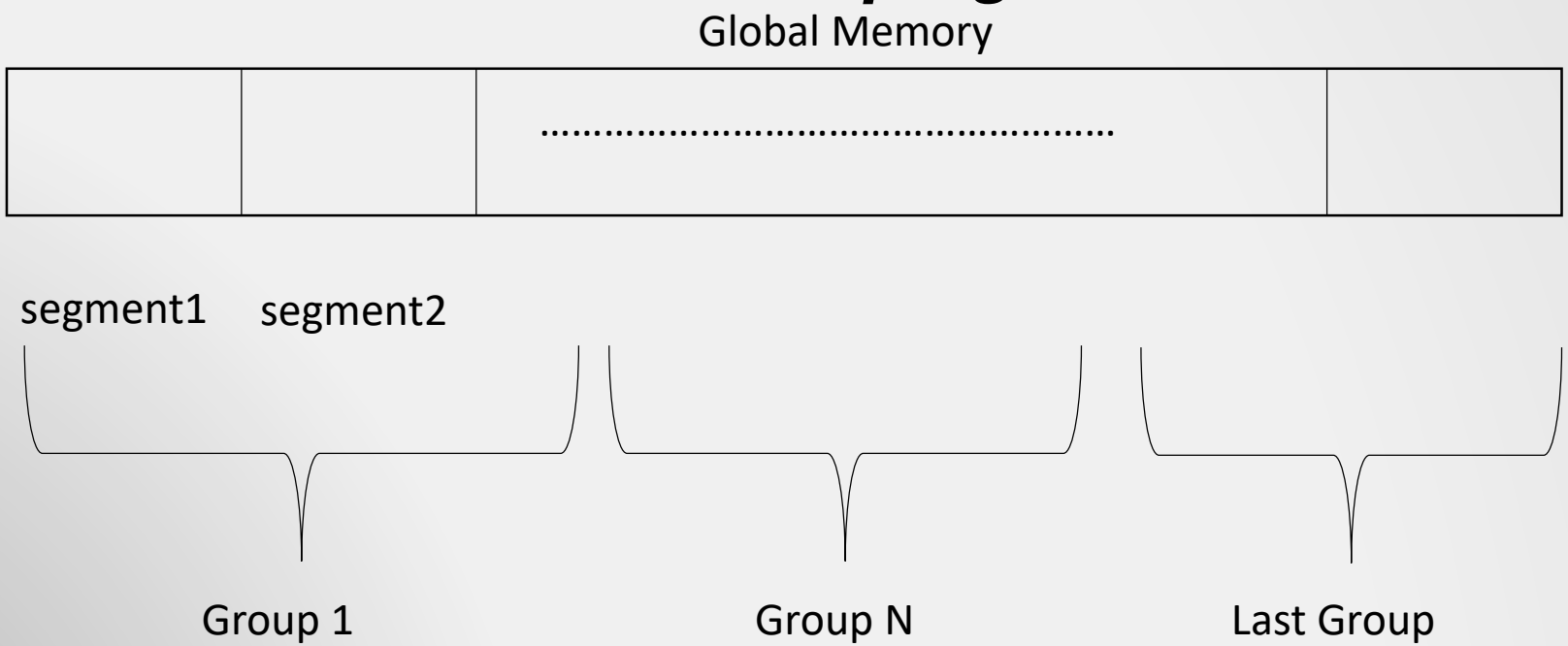
```
Exec. Time of 'full_coalesced_access' kernel = 0.00989446
Exec. Time of 'non_coalesced_access' kernel = 0.516703
Speed Up = 52.2214X
```

```
==1569== Profiling result:
==1569== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
    Kernel: full_coalesced_access(float*, float*, float*)
          1                         gld_transactions                   Global Load Transactions    26214400    26214400    26214400
          1               gld_transactions_per_request        Global Load Transactions Per Request    1.000000    1.000000    1.000000
    Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
          1                         gld_transactions                   Global Load Transactions   839546946   839546946   839546946
          1               gld_transactions_per_request        Global Load Transactions Per Request   31.093373   31.093373   31.093373
```

Performance Optimization and Efficiency

Ozcan Dulger, NCC Turkey

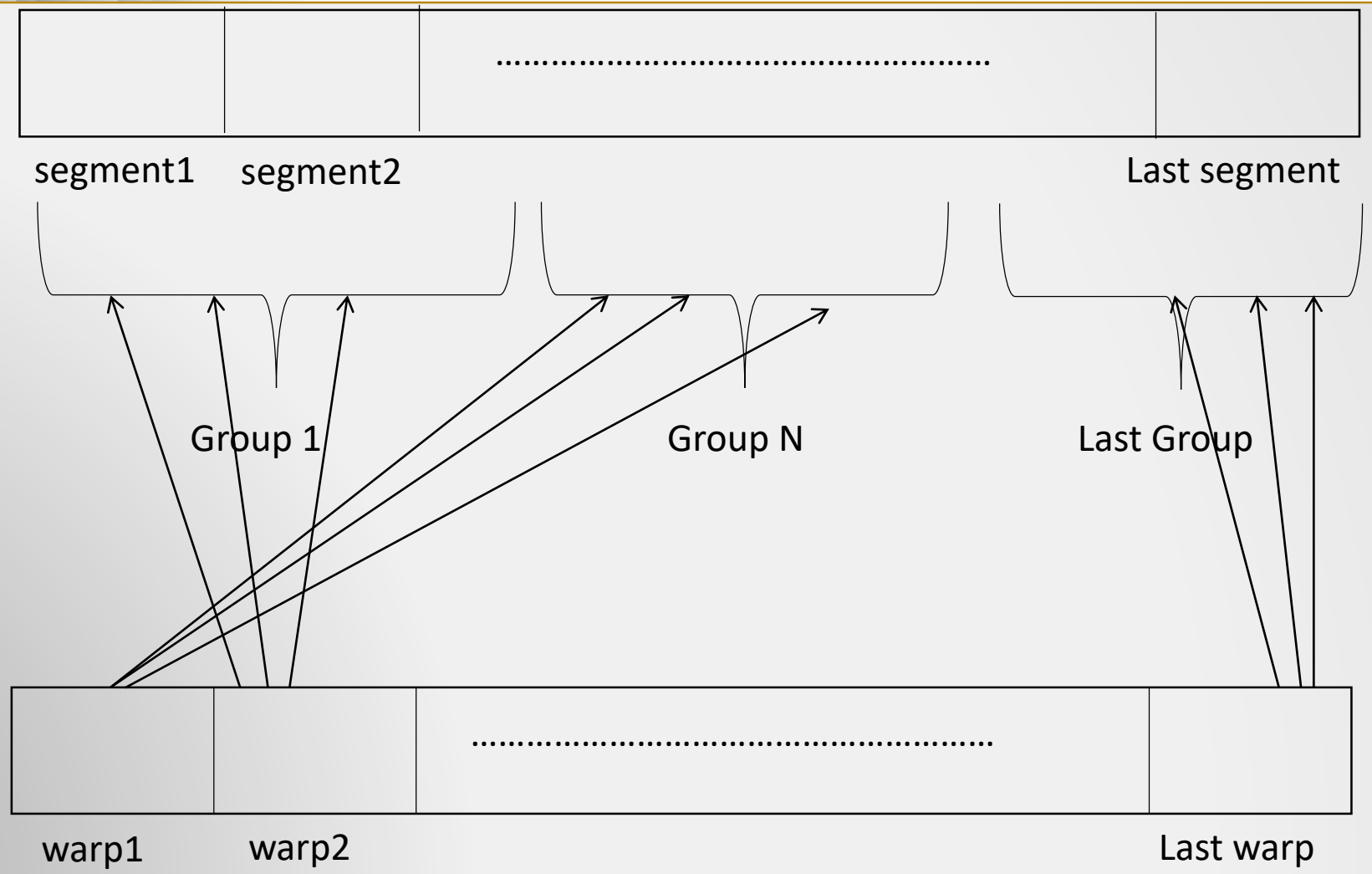
Grouping



- A group consists of contiguous segments
- The number of segments in a group can be
  - between 1 and data_size/32 (32 is the number of data in a segment)

Ref: Dülger, Ö., Oğuztüzün, H. & Demirekler, M. Memory Coalescing Implementation of Metropolis Resampling on Graphics Processing Unit. J Sign Process Syst 90, 433–447 (2018)

Grouping


Global Memory
segment1
segment2
Last segment
Group 1
Group N
Last Group
warp1
warp2
Last warp

```
__global__ void semi_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs1,curandInitializer RNGs2,unsigned int NPP_group_count,unsigned int NPP_group_size,unsigned int
GS)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state1,state2;// States of the generators
    RNGs1.load(state1,tid);// Loading the state of first generator
    RNGs2.load(state2,tid);// Loading the state of second generator

    unsigned int GN = curand(&state2)/NPP_group_count;//Generate a random number between 0 and group_count-1 (Pick a random group)
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = (curand(&state1)/NPP_group_size) + (GN*GS);//Generate a random number between 0 and group_size-1 then shift the index (Pick a random data within the selected group)
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host ;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays

    for(int counter = 0;counter < data_size; counter++)
    {
        A_host[counter] = counter+1;//Assigning numbers from 1 to size
        B_host[counter] = counter+2;//Assigning numbers from 2 to size+1
    }

    cudaMemcpy(A_GPU,A_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
    cudaMemcpy(B_GPU,B_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);

    unsigned int NTB = 1024;//Number of threads in a block
    unsigned int NP_data_size = (unsigned long int)pow(2,32)/data_size;// Number of partitions for 'data_size'

    unsigned int segment_size = 128;//Number of bytes of a segment
    unsigned int group_size = 16*(segment_size/4);//Number of data in a group
    unsigned int group_count = data_size/group_size;//Number of groups for 'data_size'
    unsigned int NP_group_count = (unsigned long int)pow(2,32)/group_count;// Number of partitions for 'group_count'
    unsigned int NP_group_size = (unsigned long int)pow(2,32)/group_size;// Number of partitions for 'group_size

    dim3 threadsPerBlock(NTB);//Number of threads in a block
    dim3 numBlocks(data_size/NTB);//Number of blocks in a grid

    full_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'full_coalesced_access' kernel
    non_coalesced_access<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,RNGs,NP_data_size);//Launching 'non_coalesced_access' kernel
    cudaDeviceSynchronize();//Waits until the kernel completes its run
}
```

- The number of segments is 16 in a group
- So the memory operations of a warp will perform at most 16 transactions

```
__global__ void semi_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs1,curandInitializer RNGs2,unsigned int NPP_group_count,unsigned int NPP_group_size,unsigned int
GS)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state1,state2;// States of the generators
    RNGs1.load(state1,tid);// Loading the state of first generator
    RNGs2.load(state2,tid);// Loading the state of second generator

    unsigned int GN = curand(&state2)/NPP_group_count;//Generate a random number between 0 and group_count-1 (Pick a random group)
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = (curand(&state1)/NPP_group_size) + (GN*GS);//Generate a random number between 0 and group_size-1 then shift the index (Pick a random data within the selected group)
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 32768;//Data size
    float *A_host,*B_host,*C_host ;//Host Arrays
```

```
Exec. Time of 'full_coalesced_access' kernel = 0.000113216
Exec. Time of 'semi_coalesced_access' kernel = 0.000788544
Speed Up = 6.96495X
Exec. Time of 'non_coalesced_access' kernel = 0.00168253
Speed Up = 14.8612X
```

```
==1748== Profiling result:
==1748== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
       Kernel: full_coalesced_access(float*, float*, float*)
         1                            gld_transactions                   Global Load Transactions      204800      204800      204800
         1                gld_transactions_per_request      Global Load Transactions Per Request    1.000000    1.000000    1.000000
       Kernel: semi_coalesced_access(float*, float*, float*, curandInitializer, curandInitializer, unsigned int, unsigned int, unsigned int)
         1                            gld_transactions                   Global Load Transactions     2870920     2870920     2870920
         1                gld_transactions_per_request      Global Load Transactions Per Request   13.414511   13.414511   13.414511
       Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
         1                            gld_transactions                   Global Load Transactions     6461782     6461782     6461782
         1                gld_transactions_per_request      Global Load Transactions Per Request   30.632689   30.632689   30.632689
```

```
__global__ void semi_coalesced_access(float *A,float *B,float *C,curandInitializer RNGs1,curandInitializer RNGs2,unsigned int NPP_group_count,unsigned int NPP_group_size,unsigned int
GS)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    curandState_t state1,state2;// States of the generators
    RNGs1.load(state1,tid);// Loading the state of first generator
    RNGs2.load(state2,tid);// Loading the state of second generator

    unsigned int GN = curand(&state2)/NPP_group_count;//Generate a random number between 0 and group_count-1 (Pick a random group)
    unsigned int index,i;

    for(i=0;i<100;i++)
    {
        index = (curand(&state1)/NPP_group_size) + (GN*GS);//Generate a random number between 0 and group_size-1 then shift the index (Pick a random data within the selected group)
        C[tid] = A[index] + B[index];//Vector addition
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host ;//Host Arrays
```

```
Exec. Time of 'full_coalesced_access' kernel = 0.00996032
Exec. Time of 'semi_coalesced_access' kernel = 0.221406
Speed Up = 22.2288X
Exec. Time of 'non_coalesced_access' kernel = 0.516618
Speed Up = 51.8676X
```

```
==2090== Profiling result:
==2090== Metric result:
Invocations                               Metric Name                        Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
       Kernel: full_coalesced_access(float*, float*, float*)
         1                         gld_transactions                   Global Load Transactions    26214400    26214400    26214400
         1               gld_transactions_per_request        Global Load Transactions Per Request    1.000000    1.000000    1.000000
       Kernel: semi_coalesced_access(float*, float*, float*, curandInitializer, curandInitializer, unsigned int, unsigned int, unsigned int)
         1                         gld_transactions                   Global Load Transactions   367412642   367412642   367412642
         1               gld_transactions_per_request        Global Load Transactions Per Request   13.412134   13.412134   13.412134
       Kernel: non_coalesced_access(float*, float*, float*, curandInitializer, unsigned int)
         1                         gld_transactions                   Global Load Transactions   839548386   839548386   839548386
         1               gld_transactions_per_request        Global Load Transactions Per Request   31.093427   31.093427   31.093427
```

Ozcan Dulger, NCC Turkey

EURO

Occupancy

- is the ratio of active warps to the maximum number of resident warps supported on a multiprocessor
- is related with resource limitations of the SMX. These limitations are:
  - Maximum number of threads per multiprocessor (2048)
  - Maximum number of threads per block (1024)
  - Maximum number of blocks per multiprocessor (16)
  - Shared memory and registers
    - --ptxas-options=-v gives us the shared memory and register usage
- The main target is to find the optimum number of threads in a block in order to achieve maximum occupancy

Ozcan Dulger, NCC Turkey

EURO

Occupancy

- Set the block size as 64:
  - At most 16x64 (1024) threads can be active in a SMX
  - %50 theoretical occupancy
- Set the block size as 128:
  - At most 16x128 (2048) threads can be active in a SMX
  - %100 theoretical occupancy
- Set the block size as 1024:
  - At most 2x1024 (2048) threads can be active in a SMX
  - %100 theoretical occupancy

```cpp
__global__ void vector_add(float *A,float *B,float *C)
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    for(int i=0;i<1000000;i++)
    {
        if( (tid/32) % 4 == 0)
            C[tid] = A[tid] + B[tid];//Vector addition
        else if( (tid/32) % 4 == 1)
            C[tid] = A[tid] - B[tid];//Vector subtraction
        else if( (tid/32) % 4 == 2)
            C[tid] = A[tid] * B[tid];//Vector multiplication
        else if( (tid/32) % 4 == 3)
            C[tid] = A[tid] / B[tid];//Vector division
    }
}

int main(int argc, char **argv)
{
    int data_size;//Data size

    float *A_host,*B_host,*C_host;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays

    int NTB;//Number of threads per block
    if(data_size <= 1024)//Scenario 1 - Set NTB to data_size until 2^
        NTB = data_size;
    else
        NTB = 1024;
    dim3 threadsPerBlockS1(NTB);//Number of threads in a block
    dim3 numBlocksS1(data_size/NTB);//Number of blocks in a grid

    vector_add<<<numBlocksS1,threadsPerBlockS1>>>(A_GPU,B_GPU,C_GPU);

    if(data_size <= 512)//Scenario 2 - Increase NTB to its double
        NTB = 32;
    else if(data_size == 1024)
        NTB = 64;
    else if(data_size == 2048)
        NTB = 128;
    else if(data_size == 4096)
        NTB = 256;
    else if(data_size == 8192)
        NTB = 512;
    else
        NTB = 1024;
    dim3 threadsPerBlockS2(NTB);//Number of threads in a block
    dim3 numBlocksS2(data_size/NTB);//Number of blocks in a grid

    vector_add<<<numBlocksS2,threadsPerBlockS2>>>(A_GPU,B_GPU,C_GPU);

}
```

- We set the number of iterations as 1000000 so that ‘if-elseif’ structure dominates the execution time
- No warp divergence is occurred
- We use **cudaEventRecord** in order to measure the kernel execution times

In scenario 1, we set the number of threads to data_size until data_size becomes 2048

In scenario 2, we set the number of threads to 32 until data_size becomes 1024. Then we double the number of threads until data_size becomes 32768

Performance Optimization and Efficiency

Ozcan Dulger, NCC Turkey

EURO

Occupancy

data_size	Scenario 1			Scenario 2		
	# of threads	# of blocks	T. Occup.	# of threads	# of blocks	T. Occup.
32	32	1	%25	32	1	%25
64	64	1	%50	32	2	%25
128	128	1	%100	32	4	%25
256	256	1	%100	32	8	%25
512	512	1	%100	32	16	%25
1024	1024	1	%100	64	16	%50
2048	1024	2	%100	128	16	%100
4096	1024	4	%100	256	16	%100
8192	1024	8	%100	512	16	%100
16384	1024	16	%100	1024	16	%100
32768	1024	32	%100	1024	32	%100
65536	1024	64	%100	1024	64	%100

- In the first scenario, we try to increase the theoretical occupancy
- In the second scenario, we try to distribute the blocks to the SMs evenly in order to utilize the SMs efficiently

SM Efficiency

- **sm_efficiency** metric: The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU
  - First, the ratios of the running time of each SM to the total running time of the GPU is calculated. Then, the average of these ratios is the result of the metric
- **achieved_occupancy** metric: The ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
  - achieved_occupancy can not exceed the theoretical occupancy

Data	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2	S1	S2
Size	# of threads		# of blocks		SM Efficiency		T. Occup.		Achieved Occ.		Exec. Time	
32	32	32	1	1	6.54%	6.55%	25%	25%	1.5%	1.5%	0.35	0.35
64	64	32	1	2	6.54%	12.46%	50%	25%	2.9%	1.5%	0.41	0.41
128	128	32	1	4	6.54%	18.83%	100%	25%	4.3%	1.5%	0.69	0.69
256	256	32	1	8	6.54%	37.17%	100%	25%	8.8%	1.5%	0.70	0.69
512	512	32	1	16	6.54%	68.48%	100%	25%	17%	1.6%	0.71	0.69
1024	1024	64	1	16	6.54%	78.32%	100%	50%	35%	2.8%	0.74	0.70
2048	1024	128	2	16	13.15%	96.32%	100%	100%	35%	4.7%	0.75	0.70
4096	1024	256	4	16	26.26%	95.91%	100%	100%	35%	9.4%	0.75	0.72
8192	1024	512	8	16	52.41%	95.10%	100%	100%	35%	18%	0.75	0.74
16384	1024	1024	16	16	83.29%	83.28%	100%	100%	39%	39%	0.91	0.91
32768	1024	1024	32	32	62.07%	62.11%	100%	100%	72%	72%	1.6	1.6
65536	1024	1024	64	64	72.80%	72.80%	100%	100%	77%	75%	2.49	2.49

- Values with green background mean the scenario is better than the other scenario for the corresponding metric
- Values with yellow background mean both scenarios have the same values of parameters. Hence the values of the outputs are almost same
- Having better theoretical and achieved occupancy does not always mean better execution time performance
  - In this example, SM efficiency is more effective on the execution time of the kernel
  - Although S1 has better occupancy, the execution times of S2 are better than those in S1 in some of the cases

Ozcan Dulger, NCC Turkey

EURO

Causes of Low Achieved Occupancy

1. Unbalanced workload within blocks
   - the warps in a block have unbalanced workload
2. Unbalanced workload across blocks
   - the blocks in a grid have unbalanced workload
3. Too few blocks launched
   - running few blocks in an SM than the maximum active blocks per SM
4. Partial last wave
   - maximum number of warps that can be active at once in an SM

- https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

Ozcan Dulger, NCC Turkey

EURO

Warp Divergence

- Some of the structures such as ‘If-Else’ structure are considered as a single instruction for a warp
- A warp completes ‘If-Else’ instruction when all the threads in the warp complete ‘If-Else’ instruction
- If the threads in a warp execute the different paths of ‘If-Else’ structure, executing these paths becomes serial
- if(tid %2 == 0)//tid is global thread id

  ……..

  else

  ……..
- First the threads with even thread id in a warp execute ‘if’ path, and the remaining threads wait
- Then the threads with odd thread id in a warp execute the ‘else’ path, and the remaining threads wait

Ozcan Dulger, NCC Turkey

EURO

Warp Divergence

- It is important that all the threads in a warp execute the same path of ‘if-Else’ structure
- This can be ensured by using warp id in the condition of the structure
- if((tid/32) %2 == 0)//tid is the global thread id

  ……..

  else

  ……..
- The threads whose warp id is even execute the ‘if’ path
- The threads whose warp id is odd execute the ‘else’ path
- So executing the paths does not become serial

```cpp
__global__ void warp_no_divergence(float *A,float *B,float *C)//No branching for the warps in 'if-elseif' structure
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    for(unsigned int i=0;i<100;i++)
    {
        if( (tid/32) % 4 == 0)
            C[tid] = A[tid] + B[tid];//Vector addition
        else if( (tid/32) % 4 == 1)
            C[tid] = A[tid] - B[tid];//Vector subtraction
        else if( (tid/32) % 4 == 2)
            C[tid] = A[tid] * B[tid];//Vector multiplication
        else if( (tid/32) % 4 == 3)
            C[tid] = A[tid] / B[tid];//Vector division
    }
}

__global__ void warp_divergence(float *A,float *B,float *C)//Four different paths for the warps in 'if-elseif' structure
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    for(unsigned int i=0;i<100;i++)
    {
        if( tid % 4 == 0)
            C[tid] = A[tid] + B[tid];//Vector addition
        else if( tid % 4 == 1)
            C[tid] = A[tid] - B[tid];//Vector subtraction
        else if( tid % 4 == 2)
            C[tid] = A[tid] * B[tid];//Vector multiplication
        else if( tid % 4 == 3)
            C[tid] = A[tid] / B[tid];//Vector division
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host ;//Host Arrays
    float *A_GPU,*B_GPU,*C_GPU;//Device Arrays

    cudaMemcpy(A_GPU,A_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);
    cudaMemcpy(B_GPU,B_host,sizeof(float)*data_size,cudaMemcpyHostToDevice);

    unsigned int NTB = 1024;//Number of threads in a block
    dim3 threadsPerBlock(NTB);//Number of threads in a block
    dim3 numBlocks(data_size/NTB);//Number of blocks in a grid

    warp_no_divergence<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'warp_no_divergence' kernel
    warp_divergence<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU);//Launching 'warp_divergence' kernel
    cudaDeviceSynchronize();//Waits until vector_add kernel completes its run
}
```

- Distribute the paths according to warp id
- First warp executes addition, second warp executes subtraction and so on

- Distribute the paths according to global thread id
- First thread executes addition, second thread executes subtraction and so on

- Sufficiently number of paths (4)
- Sufficiently number of repetitions of the instruction (100)
- Memory operations are coalesced, so the divergence dominates the execution time
- We use **cudaEventRecord** in order to measure the kernel execution times

```cpp
__global__ void warp_no_divergence(float *A,float *B,float *C)//No branching for the warps in 'if-elseif' structure
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    for(unsigned int i=0;i<100;i++)
    {
        if( (tid/32) % 4 == 0)
            C[tid] = A[tid] + B[tid];//Vector addition
        else if( (tid/32) % 4 == 1)
            C[tid] = A[tid] - B[tid];//Vector subtraction
        else if( (tid/32) % 4 == 2)
            C[tid] = A[tid] * B[tid];//Vector multiplication
        else if( (tid/32) % 4 == 3)
            C[tid] = A[tid] / B[tid];//Vector division
    }
}

__global__ void warp_divergence(float *A,float *B,float *C)//Four different paths for the warps in 'if-elseif' structure
{
    unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id

    for(unsigned int i=0;i<100;i++)
    {
        if( tid % 4 == 0)
            C[tid] = A[tid] + B[tid];//Vector addition
        else if( tid % 4 == 1)
            C[tid] = A[tid] - B[tid];//Vector subtraction
        else if( tid % 4 == 2)
            C[tid] = A[tid] * B[tid];//Vector multiplication
        else if( tid % 4 == 3)
            C[tid] = A[tid] / B[tid];//Vector division
    }
}

int main(int argc, char **argv)
{
    unsigned int data_size = 4194304;//Data size
    float *A_host,*B_host,*C_host ;//Host Arrays
```

```
Exec. Time of 'warp_no_divergence' kernel = 0.0124316
Exec. Time of 'warp_divergence' kernel = 0.0385476
Speed Up = 3.10078X
```

```
==23306== Profiling result:
==23306== Metric result:
Invocations                               Metric Name                    Metric Description         Min         Max         Avg
Device "Tesla K40c (0)"
        Kernel: warp_divergence(float*, float*, float*)
          1                 warp_execution_efficiency              Warp Execution Efficiency      34.80%      34.80%      34.80%
        Kernel: warp_no_divergence(float*, float*, float*)
          1                 warp_execution_efficiency              Warp Execution Efficiency     100.00%     100.00%     100.00%
```

Metric:

warp_execution_efficiency: Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage

References

- https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf
- https://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf
- https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html
- https://docs.nvidia.com/cuda/profiler-users-guide
- Dülger, Ö., Oğuztüzün, H. & Demirekler, M. Memory Coalescing Implementation of Metropolis Resampling on Graphics Processing Unit. J Sign Process Syst 90, 433–447 (2018) **https://rdcu.be/cLz8N**
- **https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html**
- **https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm**

Thanks!

EuroHPC
Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro

Multi-GPU and multi-stream programming

Programming on Accelerators

EuroCC / Castiel WP2 workshop

May 23th, 2022

Luca Ferraro (l.ferraro@cineca.it)
HPC Department - CINECA
EuroCC Italy

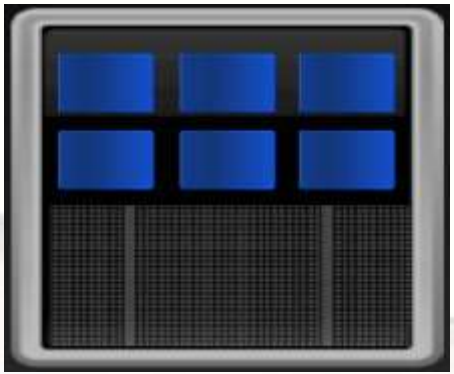
- Brief Recup of GPGPU Programming Model
- Synchronous and Asynchronous Operations
- Streams
- Concurrent Execution
- Managing multi-devices
- inter GPU communications

GPGPU Programming Model

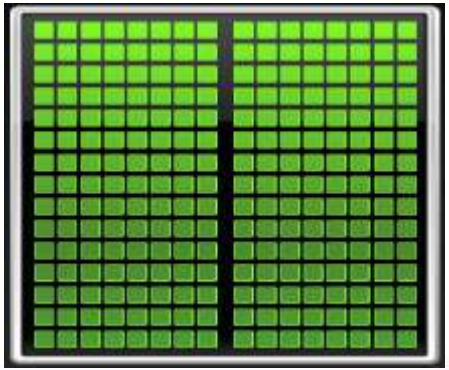
- Optimized for low-latency accesses to cached data sets
- Control logic for out-of-order and speculative execution
- Best for serial or event driven

- Optimized for data-parallel, throughput computation
- can handle thousands of threads efficiently
- Best for data-parallel tasks

CPU



GPU

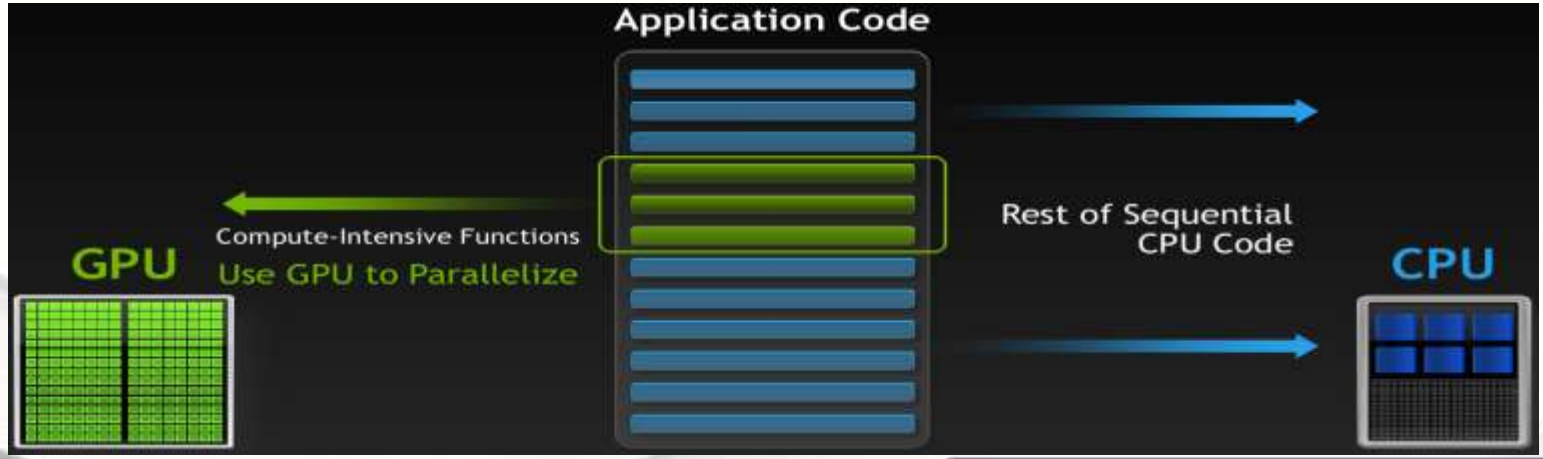


GPGPU Programming Model

- General Purpose GPU Programming relates to use of GPU computational power to solve problems other than graphics
- CPU and GPU are **separate devices** with **separate memory** space addresses
- GPU is seen as an auxilirary coprocessor equiped with thousands of cores and a high bandwidth memory
- They should work togheter for best benefit and performances

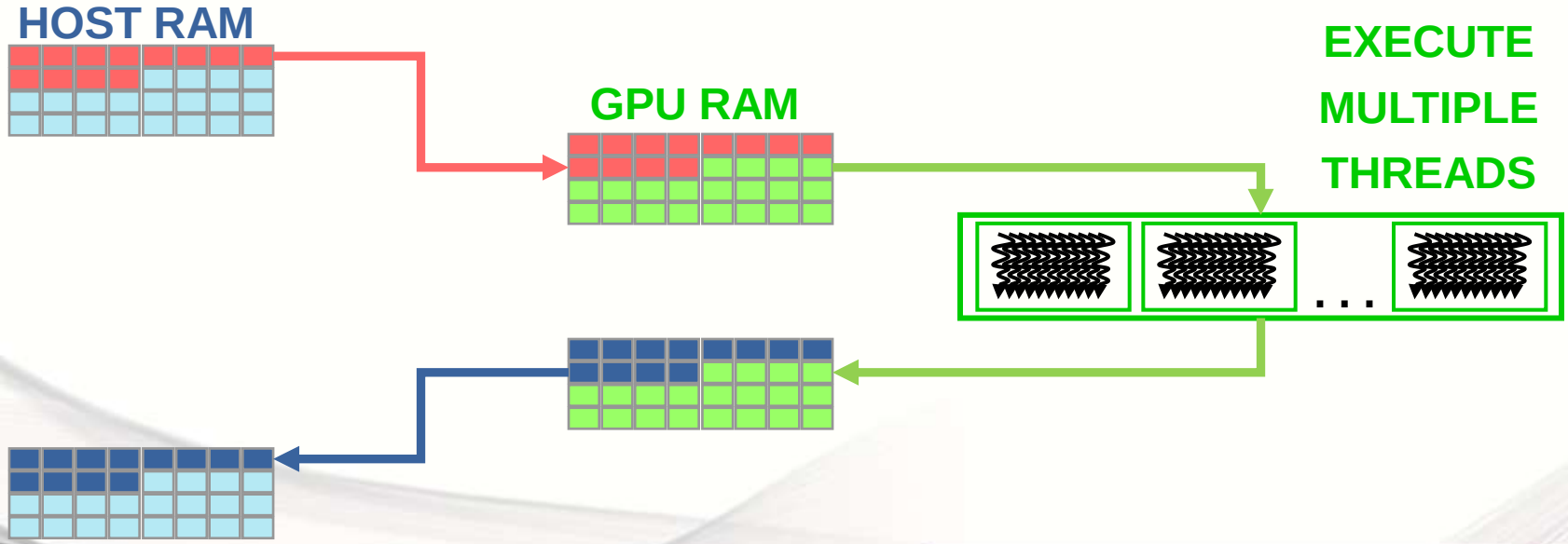
GPGPU Programming Model

- **serial parts** of a program, or those with low level of parallelism, keep running **on the CPU** (host)
- computational-intensive **data-parallel** regions are executed **on the GPU** (device)
- required data is moved on GPU memory and back to HOST memory



Data movement

- data must be moved from HOST to DEVICE memory in order to be processed on the GPU
- when data is processed, and no more needed on the GPU, it is transferred back to HOST



The data movement bottleneck

- Data movement is often *the bottleneck* of many GPU porting activities or applications
  - many unexperienced GPU developer don't keep the data transfer problem seriously enough or simply ignore it
    - some GPU paradigms/solution "hides" or automate transfers, but the driver or the compiler could make wrong choices
  - the bus transfer can be quite slow with respect to the GPU throughput capacity
    - PCIe v3 provides 12-14GB/s average transfer rate
  - sometimes data transfer can take more than the GPU computation if the problem is too "easy"
    - that's way we stressed that GPU are best suited for computational intensive problems, not just "parallel"
    - for example:
      - a vector add is not suited for GPU (order N)
      - matrix matrix multiplication is a good candidate for GPU (order N^3)

- Synchronous and Asynchronous Operations

Blocking and Non-blocking Functions

- CUDA runtime functions can be divided in two categories:

- **blocking** (synchronous):
  returns control to *host* thread after execution is completed on *device*
  - all memory transfer > 64KB
  - all memory allocation on *device*
  - allocation of page locked memory on *host*

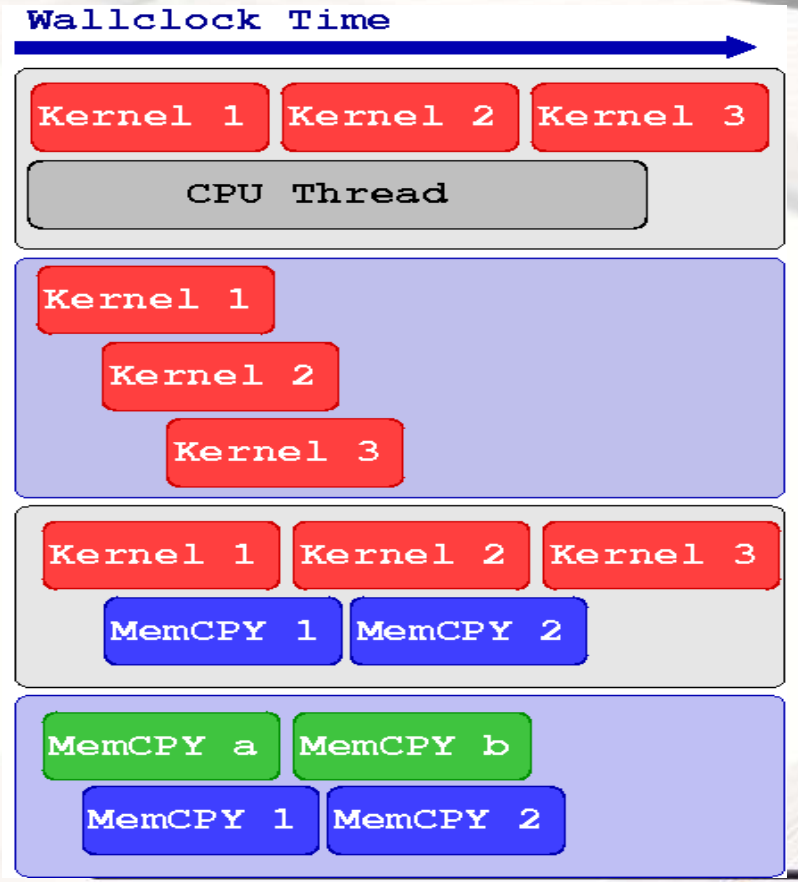
- **Non-blocking** (asynchronous):
  returns control to *host* immediatelly, while its execution proceeds on *device*
  - all kernel launches are asynchronous
  - all memory transfers < 64KB
  - memory initialization on *device* (cudaMemset)
  - memory copies from *device* to *device*
  - explicit asynchronous memory transfers

- CUDA API provides asynchronous versions of their counterpart synchronous functions

Concurrent and Asynchronous Execution

Asynchronous functions allow to perform concurrent execution:

1. Overlap computation on host and on device
2. execution of more than on kernel on the same *device*
3. data transfers between *host* and *device* while executing a kernel
4. data transfers from *host* to *device*, while transfering data from *device* to *host*



Example of Devie/Host Concurrent Execution

```
kernel <<<threads, Blocks>>> (a, b, c) // asynchronous / non-blocking call

// execute some work on CPU while GPU keeps on computing
CPU_Function()

// blocks CPU until GPU has finished its work
cudaDeviceSynchronize()

// CPU can use data resulting from the GPU computation
CPU_uses_the_GPU_kernel_results()
```

Since CUDA kernel invocation is an asynchronous operation, CPU can proceed and evalutate the **`CPU_Function()`** while the GPU is involved in kernel execution (*concurrent execution*).

Before using the results from you CUDA kernel, some form of synchronization between *host* and *device* is required.

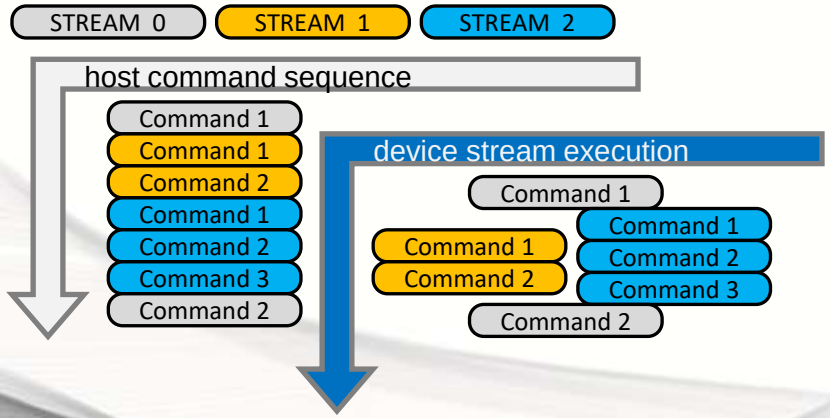
- Streams

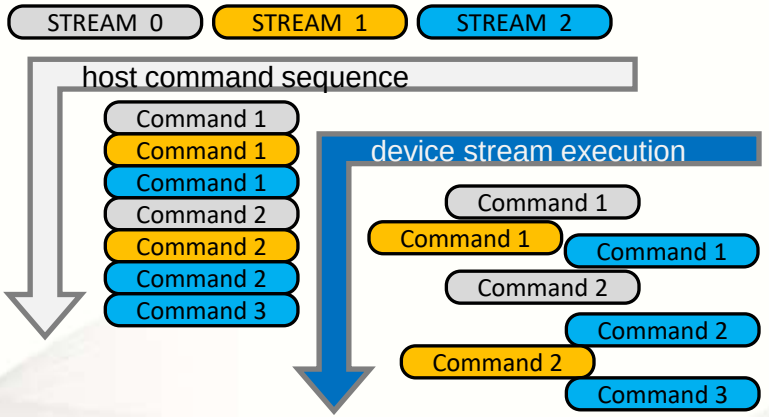
CUDA Streams

- GPU operations are implementated in CUDA using execution queues, called **streams**
- any operation pushed in a stream will be executed only after all other operations in the same stream are completed
  - FIFO queue behaviour
- operations assigned to different streams can be executed in any order with respect to each other
- The CUDA runtime provides a **default stream** (stream 0) which will be the default queue of all operation if not explicitly declared otherwise

CUDA Streams

- All operations assigned to the **default stream** will be executed only after all preceeding operations already assigned to other streams are completed
- Any further operation assigned to other stream different from the default will begin only after all operations on the default stream are completed
- operations assigned to the default stream act as implicit synchronization barriers among other streams
- remeber: operations assigned to different streams can be executed with any precedence with respect other streams





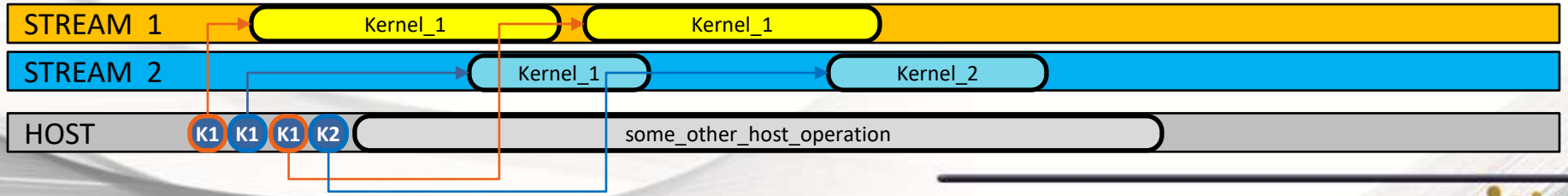
Kernel Concurrent Execution

```
cudaStream_t stream1, stream2;

cudaStreamCreate(stream1);
cudaStreamCreate(stream2);

// concurrent launch of the same kernel on different data
Kernel_1<<<blocks, threads, shmem_size, stream1>>>(inp_1, out_1);
Kernel_1<<<blocks, threads, shmem_size, stream2>>>(inp_2, out_2);

// concurrent launch of different kernels
Kernel_1<<<blocks, threads, shmem_size, stream1>>>(inp, out_1);
Kernel_2<<<blocks, threads, shmem_size, stream2>>>(inp, out_2);
some_other_host_operation();

cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```

potentially overlapped !

potentially overlapped !

overlapped also with host !



Synchronization

- **Explicit Synchronizations** :
  - `cudaDeviceSynchronize()`
    - Blocks host code until all operations on the device are completed
  - `cudaStreamSynchronize(stream)`
    - Blocks host code until all operations on a stream are completed
  - `cudaStreamWaitEvent(stream, event)`
    - Blocks all operations assigned to a stream until event is reached
- **Implicit Synchronizations** :
  - All operations assigned to the default stream
  - All page-locked memory allocations
  - All memory allocations on device
  - All settings operation on device
  - …

- Asynchronous Data Transfers

Asynchronous Data Transfers

- *host* memory must be of page-locked type (a.k.a pinned) in order to perform asynchronous data transfers between host and device
- CUDA runtime provides the following functions to handle page-locked memory:
  - `cudaMallocHost()` allocate page-locked memory on *host*
  - `cudaFreeHost()` free page-locked allocated memory
  - `cudaHostRegister()` turn *host* allocated memory into page-locked
  - `cudaHostUnregister()` turn page-locked memory into ordinary memory
- the `cudaMemcpyAsync()` function explicitly performs asynchronous data transfers between *host* and *device* memory
- data transfer operations should be queued into a stream different from the default one in order to be asynchronous
- Using page-locked memory allows data transfers between *host* and *device* memory with higher bandwidth performances

Asynchronous Data Transfers

```
// pseudo-code to illustrate CUDA asynchronous data transfers

cudaStreamCreate(stream_a)
cudaStreamCreate(stream_b)

cudaMallocHost(h_buffer_a, buffer_a_size)
cudaMallocHost(h_buffer_b, buffer_b_size)

cudaMalloc(d_buffer_a, buffer_a_size)
cudaMalloc(d_buffer_b, buffer_b_size)

// asynchronous and concurrent data transfers H2D and D2H
cudaMemcpyAsync(d_buffer_a, h_buffer_a, buffer_a_size, cudaMemcpyHostToDevice, stream_a)
cudaMemcpyAsync(h_buffer_b, d_buffer_b, buffer_b_size, cudaMemcpyDeviceToHost, stream_b)

cudaStreamDestroy(stream_a)
cudaStreamDestroy(stream_b)

cudaFreeHost(h_buffer_a)
cudaFreeHost(h_buffer_b)
```

Using Streams for Pipelining (Chunking)

imagine you have a set of data you have to transform with a kernel:

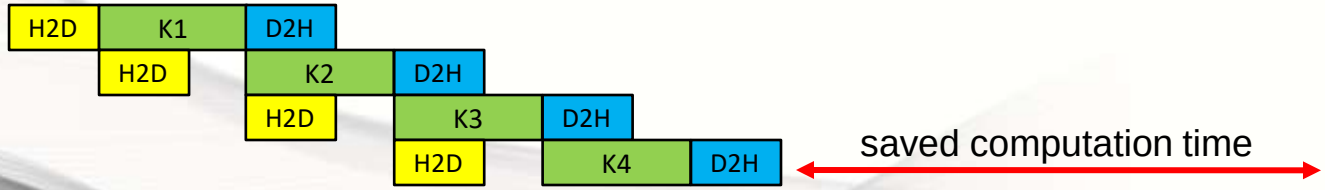
- copy data to device, launch kernel, copy results bacj to host

since we are dealing with parallel transformations we can split our data into chunks:

- final result is independent on the order in which trasnformation is applied to data

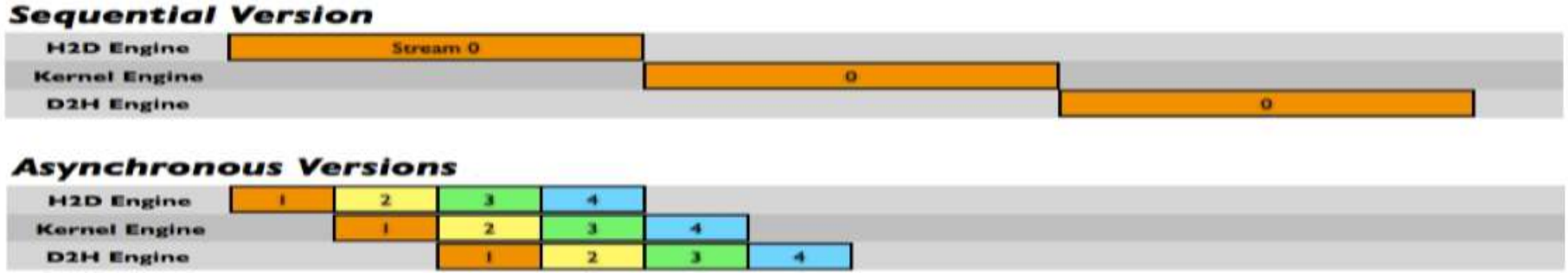
now, let's arrange chunks into a set smaller package of computation, each on a different chunk:

we can distribute these packages on different streams and perfom pipelined transformation:



Asynchronous Data Transfers

```
cudaStream_t stream[4];
for (int i=0; i<4; ++i) cudaStreamCreate(&stream[i]);

float* hPtr; cudaMallocHost((void**)&hPtr, 4 * size);

for (int i=0; i<4; ++i) {
  cudaMemcpyAsync(d_inp + i*size, hPtr + i*size,
                  size, cudaMemcpyHostToDevice, stream[i]);

  MyKernel<<<100, 512, 0, stream[i]>>>(d_out+i*size, d_inp+i*size, size);

  cudaMemcpyAsync(hPtr + i*size, d_out + i*size,
                  size, cudaMemcpyDeviceToHost, stream[i]);
```

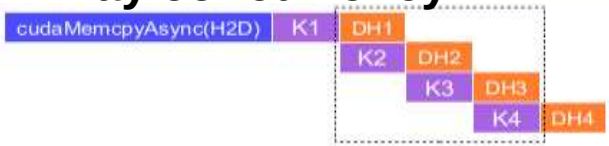


Concurrency

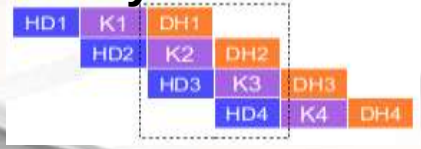
- Concurrency: when two or more CUDA operations proceed at the same time
  - from **Kepler** and higher nvidia GPU models: up to 32 way concurrency
  - 2 data transfers host/device (duplex)
  - concurrency with host operations

Serial :

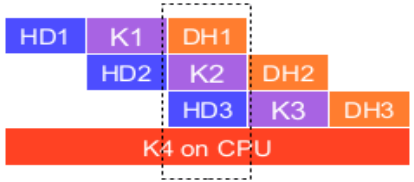
2 way concurrency :



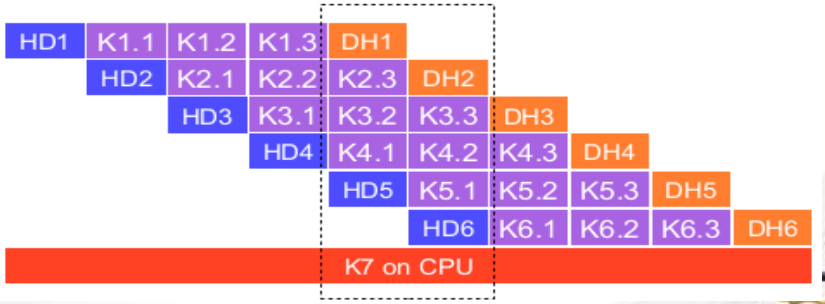
3 way concurrency :



4 way concurrency :



4/+ way concurrency :



- Multiple GPU

Device Management

CUDA runtime allows to control all GPU device available on a computing node:

- get information on available CUDA enabled devices
- get specifications of each device (capability, memory sizes, SM units, etc)
- select a device and enqueue CUDA operations on that device
- manage synchronization among streams running on available devices

```
cudaDeviceCount(&number_of_gpus);

for (int gpuid = 0; gpuid < number_of_gpus; gpuid++) {
  cudaSetDevice(gpuid);
  kernel <<<threads, Blocks>>> (a, b, c);
  // kernel launch is not blocking, so no need to use non-default streams here
}

for (int gpuid = 0; gpuid < number_of_gpus; gpuid++) {
  cudaSetDevice(gpuid);
  cudaDeviceSynchronize();
}
```

Device Chunking

Multi-GPU programming can be used to speedup computation by chunking:

- distribute num_entries of data to be processed by a kernel on available GPUs
- handle starting index and reminder properly
- allocate required data for each device ...

```
cudaGetDeviceCount(&number_of_gpus);

float *data_gpu[number_of_gpus]; // use different buffers on each GPU
size_t lower[number_of_gpus], upper[number_of_gpus], width[number_of_gpus];
cudaStream_t gpu_streams[number_of_gpus]; // create non default streams on each GPU

for (int gpuid = 0; gpuid < number_of_gpus; gpuid++) {
    cudaSetDevice(gpuid); cudaStreamCreate(&gpu_streams[gpuid]);

    lower[gpuid] = chunk_size * gpuid;
    upper[gpuid] = min(lower[gpuid] + chunk_size, num_entries); // handle reminder
    width[gpuid] = upper[gpuid] - lower[gpuid];

    cudaMalloc(&data_gpu[gpu], sizeof(float)* width[gpuid]);
}
```

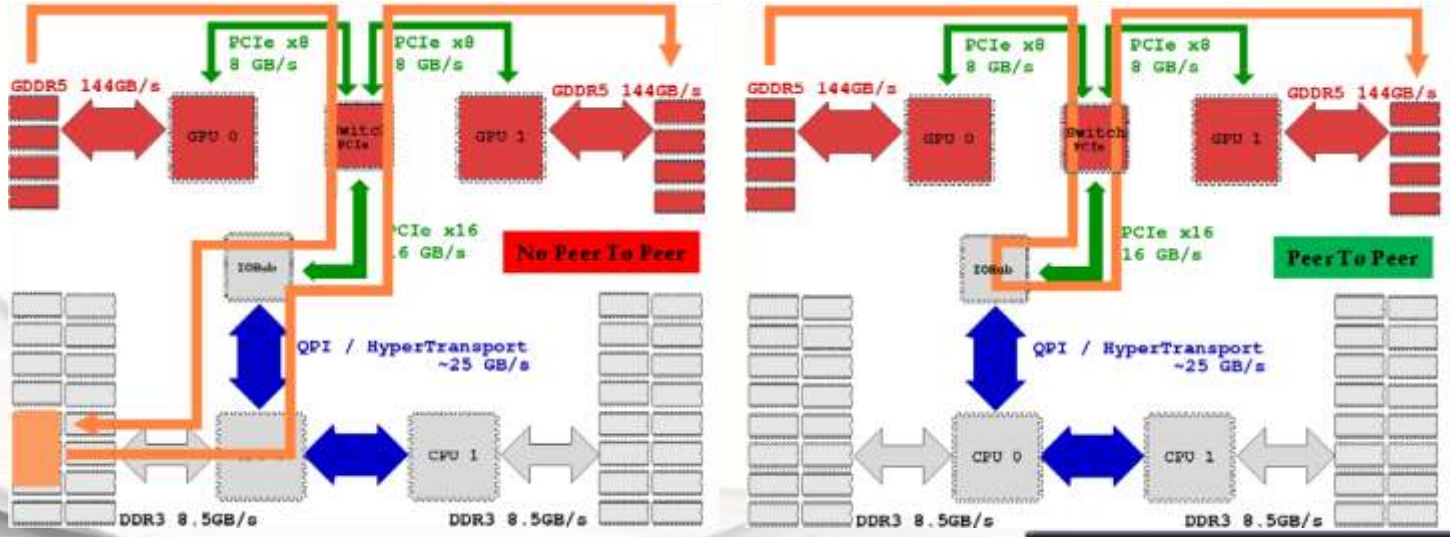
Device Chunking

- copy host data to local GPU device buffers
- launch required kernel on each device
- copy back data on host buffer
- remember: use asynchronous operations not to block host loop

```
for (int gpuid = 0; gpuid < number_of_gpus; gpuid++) {
    cudaSetDevice(gpuid);

    cudaMemcpyAsync(data_gpu[gpu], data_cpu + lower[gpuid],
           sizeof(float) * width[gpudid], cudaMemcpyHostToDevice, gpu_stream[gpuid]);

    kernel <<<grid, block, shmem, gpu_stream[gpuid]>>> (&data_gpu[gpu], width[gpuid]);

    cudaMemcpyAsync(data_cpu + lower[gpuid], data_gpu[gpu],
           sizeof(float) * width[gpudid], cudaMemcpyDeviceToHost, gpu_stream[gpuid]);
}
```

Device Communication (single-node)

- A device can directly transfer or access data to/from another device
  - This kind of direct transfer is called Peer to Peer (P2P)
- P2P transfers are more efficient and do not require buffers on host for inter-GPU exchanges
  - Direct access avoid host memory copy



Device Communication (single-node)

- P2P should be activated between two GPUs
- P2P communication availability should be queried
  - Dual-IOH systems prevent PCIe P2P across the IOH chips
  - QPI link between the IOH chips isn't compatible with PCIe P2P
  - if P2P is not available, a fall-back to D2H->H2D is automatically handled

```
// pseudo code to enable P2P communications between gpuA and gpuB

gpuA=0, gpuB=1
cudaSetDevice(gpuA)
cudaDeviceCanAccessPeer(answer, gpuA, gpuB)

If answer is true:
  cudaDeviceEnablePeerAccess(gpuB, 0)
  // gpuA performs copy from gpuA to gpuB
  cudaMemcpyPeer(buffer_B, gpuB, buffer_A, gpuA, buffer_size)
  // gpuA performs copy from gpuB to gpuA
  cudaMemcpyPeer(buffer_A, gpuA, buffer_B, gpuB, buffer_size)
```

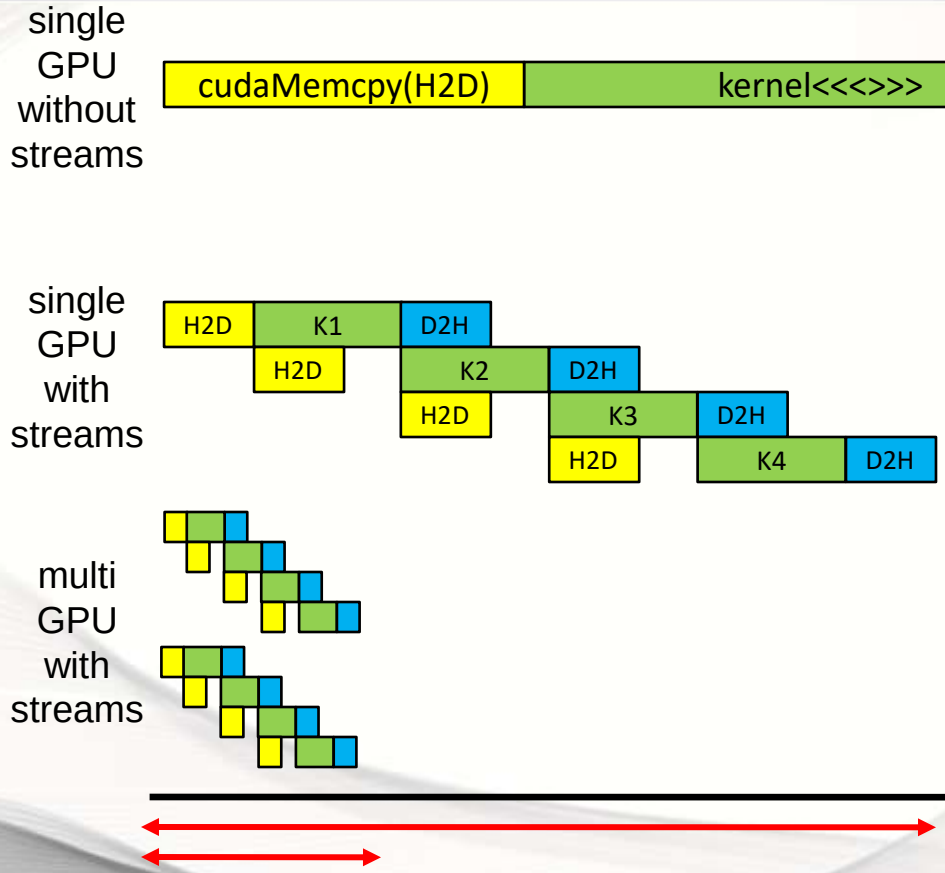
Device Communication (multi-node)

CUDA API allows to handle GPUs belonging to a single node only

- if you need to use GPUs belonging to multiple node you have to rely on other multi-precesses programming paradigms such as MPI, PGAS, etc
- there are CUDA-aware MPI implementations which allow to refer device buffers pointers as source/destination of communications (RDMA)
- other approaches are available (i.e: nvshmem) but no time to fit in this lecture

```
// common HALO EXCHANGE pattern between GPUs with traditional MPI
cudaMemcpyAsync(  ..., stream_halo[i] );  // D2H transfer
cudaStreamSynchronize( stream_halo[i] ); // be sure data is on host buffer
MPI_Sendrecv( ... );                     // perform communication (blocking)
cudaMemcpyAsync( ..., stream_halo[i] );  // H2D transfer
// repeat this for each halo side

// HALO EXCHANGE pattern with CUDA-aware
MPI_Sendrecv( field_d[left_border], ..., field_d[right_halo], ...) // send left, receive right
MPI_Sendrecv( field_d[right_border], ..., field_d[left_halo], ...) // send right, receive left
```

Wrapping Up



- we can split GPU computation into smaller chunks
- use streams and asynchronous operations to build pipe-line
- using multi-GPU to scale computation over available distributed resources

... and that's all folks !!!

Thank you for your attention
and happy programming!

EuroHPC
Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro

EUROCC ITALY

References

- on-line CUDA Programming Guide
- https://developer.nvidia.com/blog
- CUDA Streams - Best Practice and Common Pitfalls
  GTC talk by Justin Luitjens - NVIDIA
- Multi-GPU Programming Models
  GTC November 2021 by Jiri Kraus - NVIDIA

CINECA organizes courses and schools on many HPC subjects:
have a look at https://eventi.cineca.it/en/hpc for an updated list

For further questions or interest in collaboration,
please send me an email at l.ferraro@cineca.it

Rights & Credits

Slides and examples were authored by:

Luca Ferraro, Sergio Orlandini