An introduction to GPU computing with CUDA

Depto. Lenguajes y Sistemas Informáticos, Univ. Granada

Granada, 2018

Depto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Outline

Index

Graphics Processing Unit (GPU)

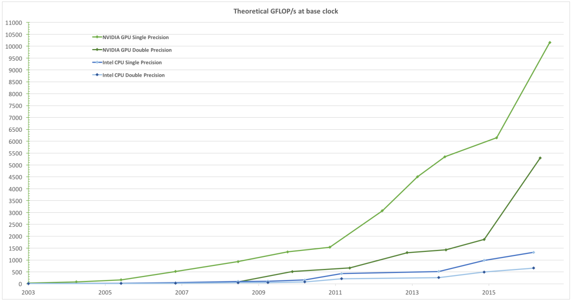
Graphics Processing Unit (GPU) Programmable single-chip processor which is used primarily for rendering of 3D graphics scenes, 3D object processing and 3D motion.

- Embedded in stand-alone cards with their **own memory** and connected to a CPU-based system (PCI-Express bus).
- **Massively parallel**: Hundreds or thousands of processing units to perform lots of arithmetic operations in parallel.
- **Highly programmable**: Non-graphics applications.
- **Excellent cost-performance ratio**

GPUs versus multicore CPUs

Large performance gap between GPUs and multicore CPUs.

- While CPUs are designed to minimize the execution latency of a single task, **GPUs are designed to maximize the total execution throughput** of a large number of parallel tasks.
- **Graphics memories** are much faster.



Source: www.nvidia.com

GPU computing

GPU computing consists of using GPUs together with CPUs to accelerate the solution of compute-intensive science, engineering and enterprise problems.

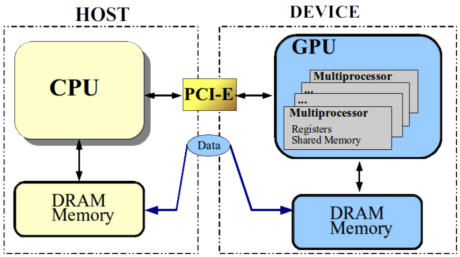
- Increasing interest in the acceleration of numerical simulations by using GPU-based computer systems.
- Initially, graphics-specific programming languages and interfaces (Cg GLSL, ...).
- NVIDIA developed the **CUDA programming toolkit** which includes an extension of the C language which facilitates the programming of GPUs.
- Additionally, **several languages and interfaces** (OpenCL, OpenACC, Thrust, ...) have been developed to make the GPU computing easier.

Index

CUDA (Compute Unified Device Architecture)

CUDA: Hardware-Software platform to exploit efficiently the potential of NVIDIA GPUs to accelerate computations.

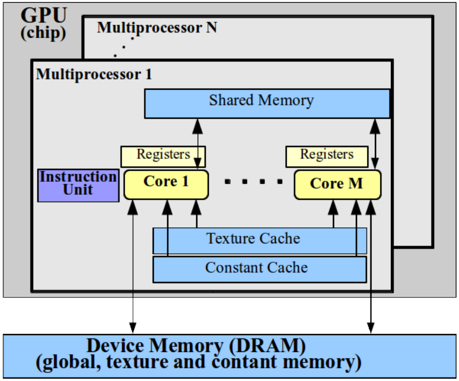
- An **Unified architectural view** of the GPU
- A **multithreaded programming model** which includes an extension to the C language (CUDA C).
- **GPU works as a coprocessor** for the main CPU (host).
- CPU system (host) and the GPU system (device) maintain their own memories: It is possible to copy data CPU-GPU and GPU-CPU.



Index

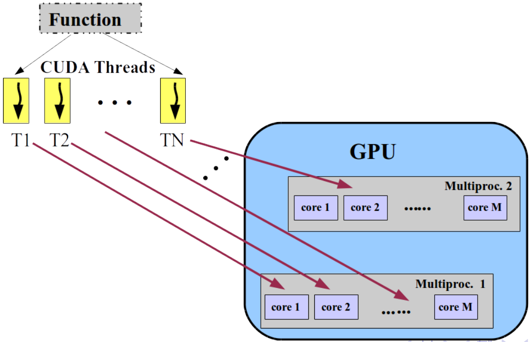
Introduction to CUDA hardware model

- A CUDA enabled GPU is formed by N multiprocessors, each one having M processors or cores (32, 192, . . .).
- At any clock cycle, each core of the multiprocessor executes the same instruction, but operates on different data.



Introduction to CUDA Threads

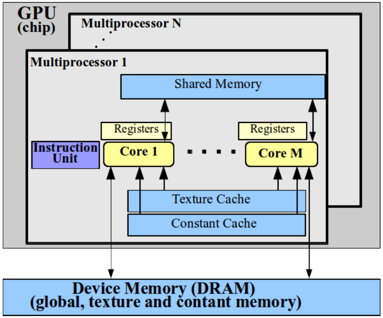
- The multiprocessors of the GPU are specialized in the parallel execution of multiple CUDA threads.
- **CUDA thread**: Sequential computational task which executes an instance of a function.
- Each CUDA thread is executed concurrently with respect to other CUDA threads (executing the same function) on the cores of the GPU multiprocessors.



CUDA memory model

On-chip memories:

- **Registers**: Each GPU has a particular number of registers per multiproc. which are split and assigned to the threads running on the multiproc.
- **Shared memory**: Readable and writeable only from the CUDA threads running on a multiproc. It is small but much faster than global memory.



Off-chip memories: Shared by all CUDA threads on the GPU.

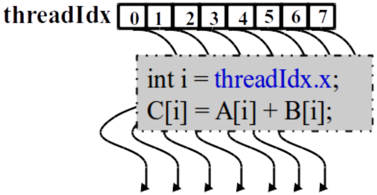
- **Global memory**: Big but slow regarding shared memory.
- **Constant and texture memories**: read-only and cached.

Index

CUDA kernels

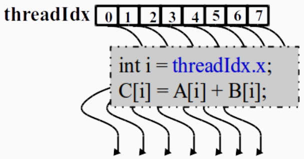
How to specify the function to be executed by each CUDA thread?

- **CUDA kernel**: special C function, called from the CPU and executed N times on the GPU by an array of N CUDA threads
- **Extremely light threads**: Recommendable to use lots of thousands of CUDA threads for high efficiency.
- Every thread executes the same code but the specific thread action depends on the **thread identifier** (built-in variable `threadIdx`).



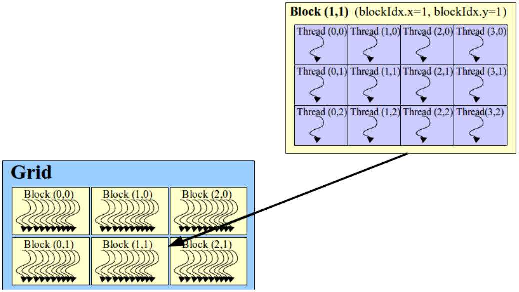
Simple CUDA kernel to add vectors

```
__global__ void VecAdd(float* A, float* B, float* C)
{  int i = threadIdx.x;
   C[i] = A[i] + B[i];  }
...
int main()
 {
    ...
    // Kernel call with N Threads
    VecAdd <<<1, N>>> (A, B, C);
 }
```



- Modifier `__global__` is used to declare a kernel function.
- Each CUDA thread computes one element of the output vector `C`.
- To launch the kernel, the programmer must specify the number of CUDA threads to execute it on the GPU (parameters between <<< and >>>).

CUDA Thread Organization. Grids and Blocks



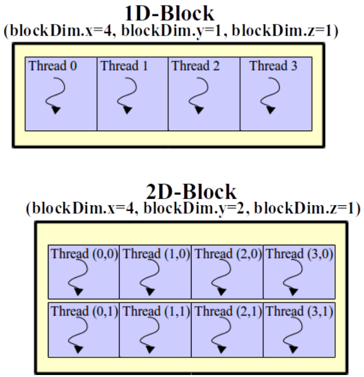
- The set of CUDA threads for a kernel launch is organized forming a **grid of thread blocks** that run concurrently.
- When launching a kernel, the number of threads per block and the number of blocks per grid are specified with this syntax:

```
<<<   blocks_per_grid, threads_per_block >>>
```

- **Simple vector-addition CUDA program**: One grid with only one block (N threads).

Blocks of Threads

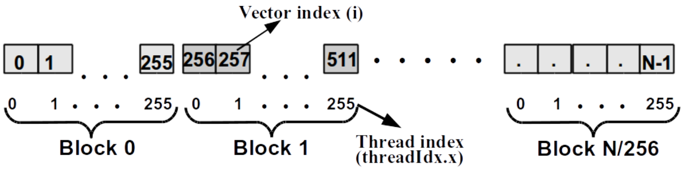
- **CUDA block**: unit used to map threads to multiprocessors.
- Each thread is identified with a (1D, 2D or 3D) thread Index (`threadIdx`) in a **thread block**.
- Each thread block runs on a single GPU multiprocessor.
- Each multiprocessor can process a maximum number of blocks and each block can include a maximum number of threads.
- `dim3 blockDim`: stores the block dimensions for a kernel.



Grids of Blocks. A general vector-addition kernel

- The blocks are organized as a (1D, 2D or 3D) matrix called **grid**.

```
#define BSIZE 256
__global__ void VecAdd(float* A, float* B, float* C, int N)
   { int i = blockIdx.x * blockDim.x + threadIdx.x;
      if (i<N)     C[i] = A[i] + B[i];
   }
...
int main()
 {
  ...
  // Kernel call with N Threads using 256 threads 1D-blocks
  VecAdd <<<ceil((float)N/BSIZE), BSIZE>>> (A,B,C,N);
  ...
 }
```



Grids of Blocks

How to identify parameters of a particular thread within a kernel function?

- **Predefined variables**
  - `uint3 blockIdx`: coordinates of a block in the grid. (`blockIdx.x`, `blockIdx.y`, `blockIdx.z`).
  - `dim3 blockDim`: dimensions of the block.
  - `dim3 gridDim`: dimensions of the grid.

```
__global__ void VecAdd(float* A,float* B,float* C,int N)
   {  int i = blockIdx.x * blockDim.x + threadIdx.x;
      if (i<N)     C[i] = A[i] + B[i];
   }
```

- Using these indices and `threadIdx`, programmer can specify what data are operated by each thread.
- Conditional `if` statement is needed to test if the global index values of a thread are within the valid range.

2D Block kernel to add matrices

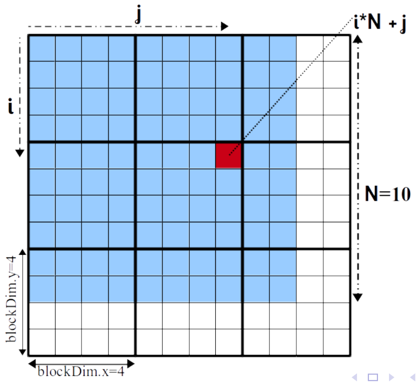
- Grid and block dimensions for a kernel must be chosen depending on the particular computation to maximize efficiency.
- **Example**:2D matrix addition, using a 2D grid of 2D-blocks.

```
...
__global__ void MatAdd (float *A, float *B, float * C, int N)
   {
    int j = blockIdx.x * blockDim.x + threadIdx.x; // Compute row index
    int i = blockIdx.y * blockDim.y + threadIdx.y; // Compute column index
    int index=i*N+j;                                         // Compute global i
    if (i < N && j < N)
         C[index] = A[index] + B[index]; // Computes C element if within range
    }
```

- Initially, compute the position of the current thread in the horizontal and vertical directions, using `blockIdx` and `threadIdx`.
- `i` and `j` are used to derive the global 1D index.
- 1D index is used to compute the corresponding element of `C` if `i` and `j`) are within the valid range.

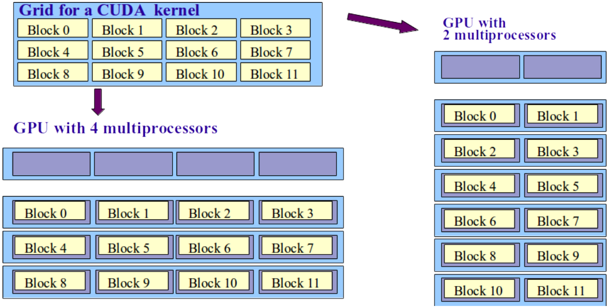
Mapping of a 10 × 10 matrix to a grid of 4 × 4 blocks

```
main()
{  .......
   // Kernel Launch code
   dim3 threadsPerBlock (4, 4);
   dim3 numBlocks( ceil ((float)(N)/threadsPerBlock.x),
                   ceil ((float)(N)/threadsPerBlock.y) );
   MatAdd <<<numBlocks, threadsPerBlock>>> (A, B, C, N);}
```

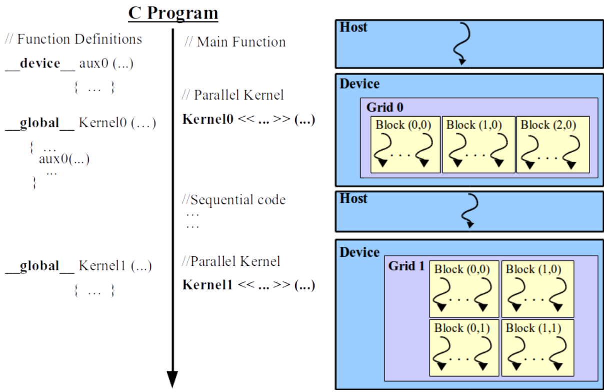


Transparent Scalability

- Execution of several blocks can follow any execution order.
- Thread blocks are automatically mapped onto multiprocessors by the CUDA runtime system
- **Transparent scalability**: The kernel execution is automatically adapted to the number of multiprocessors of the available GPU.

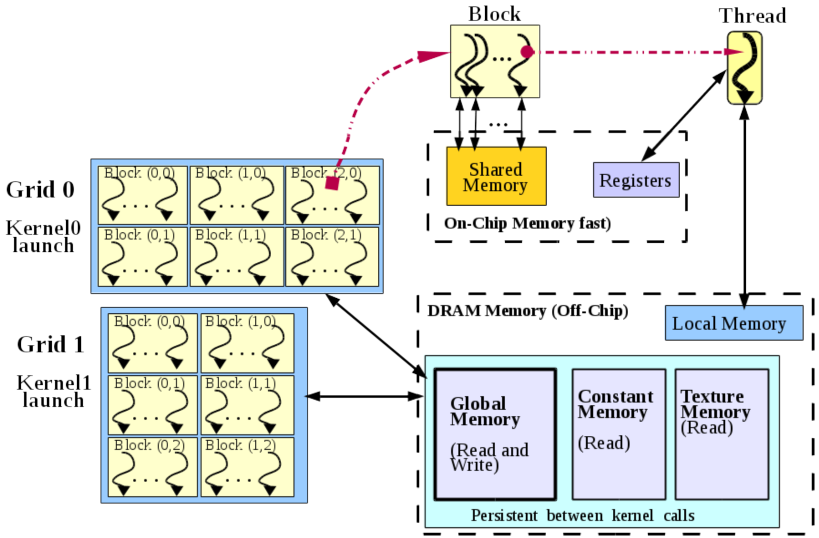


CUDA Program Structure



- A CUDA C program must include CPU code with calls to several CUDA kernels, and the specification of these kernel functions.
- A kernel can only start on a GPU when the previous CUDA function call has returned.

Memory Organization in a CUDA Program



Variables in registers and local memory

- **Registers**:
  - All scalar variables (not arrays) declared in kernel and device functions.
  - Extremely **high speed access**.
  - **Scope and lifetime**: single owner thread
  - **Max number of 32-bit registers** per thread (63-255) and multiprocessors (32768-65536) depends on the particular GPU.
- **Local memory**:
  - Array variables declared in a kernel and scalar variables which exceeding register storage capacity.
  - **Same scope and lifetime** of register variables.
  - Stored in global memory → **very low speed access**.

```
...
__global__ void example (float *input, float * output, int N)
   {int i = blockIdx.x * blockDim.x + threadIdx.x; // in register
    float w[32];                                  // in global memory
    ...      }
```

Variables in Shared memory

- **Shared memory** is part of the memory space which resides on the processor chip.
- On-chip $\rightarrow$ **very High speed access**.
- **Scope**: to read and write for all the threads within the same block
- **Lifetime**: thread block activity period.
- Mean by which **threads within a block can communicate**.
- To declare a shared memory variable in a kernel or device function, we use the qualifier `__shared__`.
  Example: Declare a shared memory vector of 32 floats in a kernel:

```
__shared__ float vector[32];
```

Dynamic arrays in Shared memory

- One can use dynamically allocated shared memory when the amount of shared memory is not known at compile time
- The shared memory allocation size per thread block must be specified using an optional execution configuration parameter. **Example**: Declare a dynamic shared memory vector of 64 floats for the kernel `reduceSum`:

```
int smemSize = 64*sizeof(float);
reduceSum<<<numBlocks, threadsPerBlock, smemSize>>>(d_V, N);
```

- In Kernel `reduceSum()` shared memory array should be declared using an unsized extern array syntax:

```
__global__ void reduceSum(float *d_V, int N)
{
      extern __shared__ float sdata[];
      ...  }
```

Multiple dynamically sized arrays in a single kernel

- Usually, one needs to declare several dynamic arrays in shared memory for a kernel.
- One must declare a single extern unsized array, and use pointers into it to divide it into multiple arrays. So for example, to declare one shared memory array of integers and another array of floats:

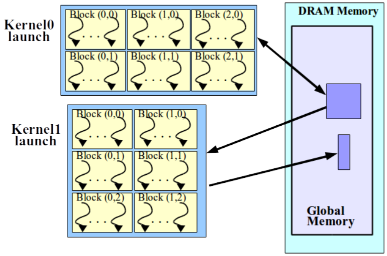
```
__global__ void transform(float *V, int N)
{
      extern __shared__ float sdata[];
      int *IntegerArray = sdata;         // N/2 integers
      float *FloatA = (float*)&integerArray[N/2]; // N floats
      ...  }
```

In the kernel launch, specify the total shared memory needed:

```
transform<<<gridSize, blockSize, N/2*sizeof(int)+N*sizeof(float))>>>(V,N);
```

Variables in Global memory

- **Main goal**: Data communication between different CUDA grids associated to different kernel calls.



- It can be written and read by the host by calling API functions.
- In DRAM memory → much lower speed access.
- **Lifetime**: Duration of the entire CUDA application
- Accessible from all threads through the entire CUDA application.
- `__device__` qualifier: declare variables in global memory.

```
__device__ float Global_A[32];
```

Pointers to Global memory

- We can declare pointers to global memory data in CUDA in a host function and then to allocate device memory to that pointer (with `cudaMalloc`).
  Example, Declare and allocate a pointer to a vector of 1024 integers in global memory:

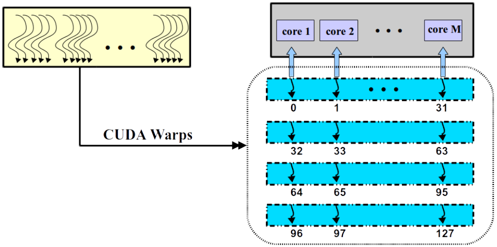
```
int numbytes = 1024*sizeof(int);
int *a_d;
cudaMalloc( (void**)&a_d, numbytes );
```

  This pointer can be passed to a kernel function as a parameter.
  Example: Parameters `A`, `B` and `C` of the `VecAdd` kernel.

Index

CUDA Thread Scheduling. Warps

- Each active block resulting from a kernel launch is divided into warps to be executed on the assigned multiprocessor.
- **Warp**: group of 32 threads with consecutive indices (ordering depends on `ThreadIdx`) that run physically in parallel on a multiprocessor.
- All threads in a warp execute the same instruction simultaneously.
- When a block is assigned to a multiprocessor, block is split into warps and a scheduler switches from one warp to another.



Decomposition of a thread block into warps

- With **1D blocks**, only `threadIdx.x` is considered:
  Example: Block size = 256 threads

First warp:	$T_0, \ldots\ldots\ldots, T_{31}$.
Second warp:	$T_{32}, \ldots\ldots\ldots, T_{63}$.
....	....
Last warp (8th):	$T_{224}, \ldots\ldots, T_{255}$.

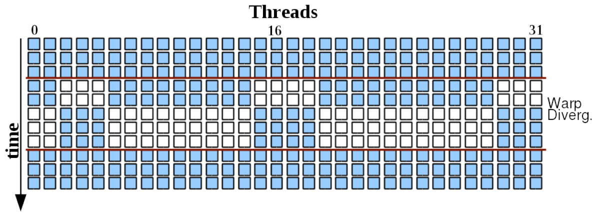
- For **2D or 3D blocks**, the dimensions are mapped to a linear 1D arrangement obtained by advancing forward firstly the x-dimension, secondly the y-dimension and finally the z-dimension.
  Example: 16×16 blocks

First warp:	$T_{0,0}, \ T_{1,0}, \ldots\ldots, T_{15,0}, \ T_{0,1}, \ T_{1,1}, \ldots\ldots, T_{15,1}$.
Second warp:	$T_{0,2}, \ T_{1,2}, \ldots\ldots, T_{15,2}, \ T_{0,3}, \ T_{1,3}, \ldots\ldots, T_{15,3}$.
....	....
Last warp (8th):	$T_{0,14}, T_{1,14}, \ldots\ldots, T_{15,14}, T_{0,15}, T_{1,15}, \ldots\ldots, T_{15,15}$.

Warp Divergence

- Unlike the CPUs, GPUs have not complex branch prediction mechanisms.
- All threads in a warp execute the same intruction on each cycle.
- **Warp Divergence**: threads in the same warp executes different instructions.

```
if (condition) {...} // condition=true for 10 threads of a warp
else {...}             // condition=false for 22 threads of the same warp
```



- **the warp executes serially each branch path** but **disabling threads that do not take that path**

Avoiding Warp Divergence

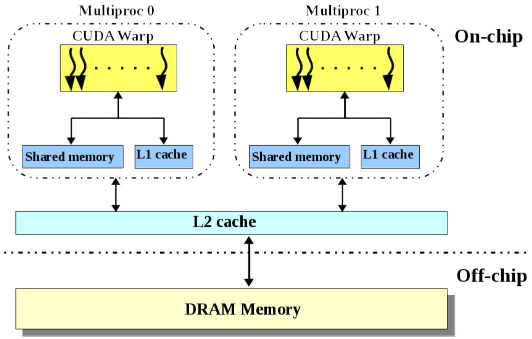
- Branch divergence occurs only within a warp.
- Avoid different execution paths within the same warp, avoiding penalty on performance.
  - **Rearrange data** in such a way as to ensure all threads in the same warp take the same path.
  - Adjust **branch granularity to be a multiple of warp size**.

```
__global__ void Kernel1(float * array) {
   int tid = blockIdx.x * blockDim.x
                            + threadIdx.x;
   float a, b;
   a = b = 0.0f;
   if (tid % 2 == 0)
             {a = 1.0f;}
   else {b = 2.0f;}
      array[tid] = a + b;
    }
```

```
__global__ void Kernel2(float * array) {
   int tid = blockIdx.x * blockDim.x
                            + threadIdx.x;
   float a, b;
   a = b = 0.0f;
   if ((tid / warpSize) % 2 == 0)
             {a = 1.0f;}
   else {b = 2.0f;}
      array[tid] = a + b;
    }
```

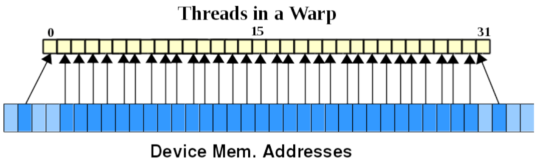
Global memory in modern GPUs

- In modern CUDA-enabled GPUs, there exists an **On-chip space memory that can be partially dedicated to cache memory** for each kernel call.
- This cache memory might reduce the global memory access cost and avoids the explicit control of the programmer.



Access Patterns to Global Memory

- Memory operations are issued per warp (32 threads).
- Threads present a single memory access (cooperatively) → One or more device memory transactions.

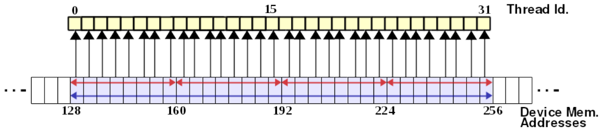


- All accesses to global memory go through L2 cache.
- Memory requests between DRAM and on-chip memory are typically served using memory transactions with:
  - **128 bytes** : if L1 and L2 caches are used.
  - **32 bytes**: when only L2 cache is used (more frequent in modern GPUs).

Types of Global memory Accesses (1)

- **Aligned memory accesses**: First address of a transaction is an even multiple of the cache granularity (32/128 bytes for L1/L2 cache).
- **Coalesced memory accesses**: all 32 threads in a warp access a contiguous chunk of memory.

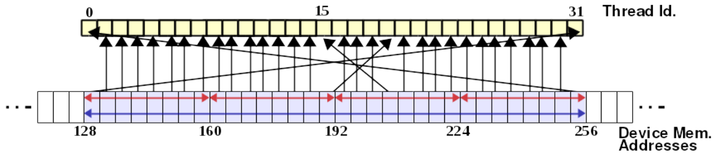
Aligned and coalesced Read Access (4 bytes/read)



- **L1 cached access**: A single 128-byte transaction is required to complete the load operation. Bus utilization = 100 %.
- **Uncached access (L2)**: Four 32-byte transactions are required. Bus utilization = 100 %.

Types of Global memory Accesses (2)

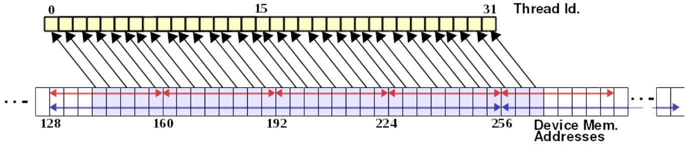
Aligned and coalesced irregular Read Access



- **L1 cached access**: A single 128-byte transaction is required . Bus utilization = 100 %.
- **Uncached access (L2)**: Four 32-byte transactions are required. Bus utilization = 100 %.

Types of Global memory Accesses (3)

Misaligned and coalesced Read Access

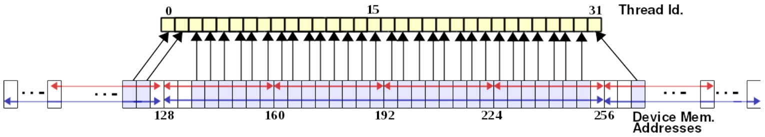


- **L1 cached access**: Two single 128-byte transactions are required. Bus utilization = 50 %.
- **Uncached access (L2)**: Five 32-byte transactions are required. Bus utilization = 80 %.

- It is very important to **organize memory operations to improve the alignment and coalescence** (to improve the global memory throughput).

Types of Global memory Accesses (4)

Misaligned and uncoalesced Read Access



- **L1 cached access**: Three single 128-byte transactions are required. Bus utilization = 33 %.
- **Uncached access (L2)**: Six 32-byte transactions are required. Bus utilization = 66 %.

- **Uncached read operations (only L2) usually lead to better bus utilization for misaligned or uncoalesced memory accesses** (fine-grained loads).

Array of structures vs. Structure of Arrays

Array of Structures (AoS)

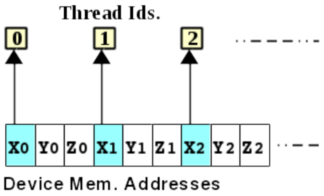
```
struct 3dpoint {
    float x;
    float y;
    float z; };
 struct 3dpoint list_of_points[N];
...

__global__ void translation
   (3dpoint *list,float d[3],
                         const int n) {
   ...
   i= blockIdx.x*blockDim.x+threadIdx.x;
   if (i < n) {
      list[i].x+=d[0];
      list[i].y+=d[1];
      list[i].z+=d[2];
   }  }
...
int main(int argc, char **argv) {
...
translation <<<grid,block>>>
    (d_list_of_points, coords, N);
...
}
```

Structure of Arrays (SoA)

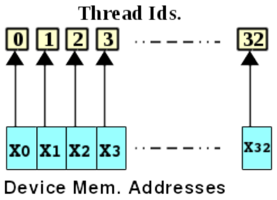
```
struct 3dpoints {
  float x[N];
  float y[N[;
  float z[N]; };
struct 3dpoints list_of_coords;
  ...
__global__ void translation
(3dpoints *lists,
              float d[3], const int n) {
    ...
    i= blockIdx.x*blockDim.x+threadIdx.x;
    if (i < n) {
      lists->x[i]+=d[0];
      lists->y[i]+=d[1];
      lists->z[i]+=d[2];
    } }
...
int main(int argc, char **argv) {
...
translation <<<grid,block>>>
     (d_list_of_coords,coords, N);
...
}
```

Array of structures vs. Structure of Arrays (2)

Array of Structures (AoS)



Structure of Arrays (SoA)



- **SoA layout** is generally more convenient in GPU.
  - Avoid interleaving of elements of the same field.
  - Item data elements that would be referenced by the same memory operation are stored adjacent to each other → **coalesced memory accesses** → a more efficient global memory utilization.

Index

CUDA C programming Interface

- **CUDA C/C++** is a minimal extension of C to define kernels as C++ functions.
- To program with a high abstraction level in CUDA C: use the **CUDA runtime API** which provides functions to perform the following tasks:
  - **Manage multiple devices**.
  - **Allocate and free device memory**.
  - **Transfer data between host memory and device memory**.
- The CUDA Runtime API requires the usage of the compiler driver **nvcc**.

CUDA Compute Capability

CUDA Compute Capability (CCC): indicates the architecture and features of the particular GPU.

- It is defined by a major revision number (core architecture) and a minor revision number `X.X`.

Features of modern GPU architectures (Fermi and Kepler)

	Fermi GF100	Fermi GF104	Kepler GK104	Kepler GK110
Compute Capability	**2.0**	**2.1**	**3.0**	**3.5**
Threads/Warp	32	32	32	32
Max Warps/Multiproc.	48	48	64	64
Max Threads/Multiproc.	1536	1536	2048	2048
Max Blocks/Multiproc.	8	8	16	16
32-bit registers/Multiproc.	32768	32768	65536	65536
Max numb. registers/Thread	63	63	63	255
Max numb. Threads/Block	1024	1024	1024	1024
Shared memory config.	16K 48K	16K 48K	16K 48K 32K	16K 48K 32K

Compiling CUDA code

- **CUDA compilation** includes **two stages**:
  - *Independent on GPU* (Virtual): It generates lower level intermediate code (PTX) (Parallel Thread eXecution).
  - *Physical*: It generates object code for a particular GPU.
- **`nvcc` tool**: decouples CPU and GPU code and performs calls to the required compilers and tools: cudac, g++,. . .
- We can generate code for devices of an specific CUDA Compute Capability by using the -gencode compiler option:
  - `-arch=<compute_xy>`: it generates PTX code for capability `x.y`.
  - `-code=<sm\_xy>`: it generate binary code for capability `x.y`, by default same as `-arch`.

Example:

```
nvcc filename.cu -gencode arch=compute_35, code=sm_35 -o filename
```

This command generates binary and PTX code compatible with CCC 3.5.

Reporting errors

- All CUDA API calls return an error code (with built-in type `cudaError_t`). This error can be originated by:
  - Error in the API call itself. For example:

```
cudaError_t err; err=cudaGetDevice(&devID);
if (err!=cudaSuccess) {cout<<"ERROR!!"<<endl;}
```

  - Error in an earlier asynchronous operation (for example in a kernel call):

```
kernel<<<blocksPerGrid,threadsPerBlock >>>(a,b,c);
err = cudaGetLastError();
if (err != cudaSuccess) {...}
```

- We can get a message which describes the particular error:

```
char *cudaGetErrorString(cudaError_t)
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Device Memory Allocation and Release

- **Allocate memory device**:
  `cudaMalloc(void ** pointer, size_t numbytes)`
- **Free memory device**: `cudaFree(void* pointer)`
- **Set a particular value in linear device memory**:
  `cudaMemset(void * pointer, int value, size_t nbytes)`

Example: Reset all elements of a vector with 1024 integers

```
int numbytes = 1024*sizeof(int);
int *a_d;
cudaMalloc( (void**)&a_d, numbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

Host-Device Data Transfers

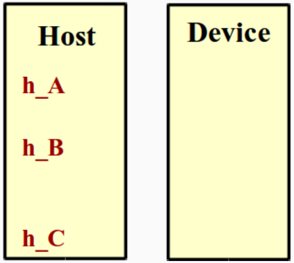
CudaMemcpy funtion is used to transfer data between global memory and host memory:

```
CudaMemcpy (void *dest, void *source, size_t nbytes,
                         enum cudaMemcpyKind direction);
```

- Copies **nbytes** bytes from the memory area pointed to by **source** to the memory area pointed to by **dest**.
- The location (host or device) for **dest**/**source** is given by **direction**:
  - **cudaMemcpyHostToDevice**: From host to device.
  - **cudaMemcpyDeviceToHost**: From device to host.
  - **cudaMemcpyDeviceToDevice**: Between positions in device.
- The invocation blocks the CPU thread and only returns when the data copy has been completed.
- Transfer does not begin until any previous CUDA calls returned.

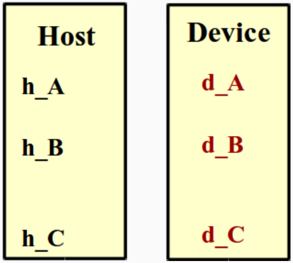
Host-Device Data Transfers. Example

```
int main()
{__global__ void VecAdd(float* A, float* B, float* C,int N){ ... }
...
int N = ...;
int size = N * sizeof(float);

float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
float* h_C = (float*) malloc(size);
Initialize(h_A, h_B, N);

float* d_A; float* d_B;float* d_C;
cudaMalloc((void**)&d_C, size);
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);

cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

VecAdd <<<(ceil((float) N/BLOCKSIZE), BLOCKSIZE>>> (d_A, d_B, d_C, N);

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

cudaFree(d_A);cudaFree(d_B); cudaFree(d_C);}
```



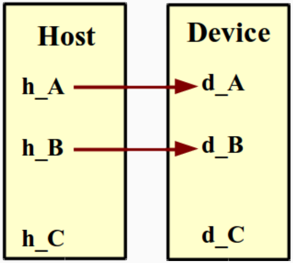
Host-Device Data Transfers. Example

```
int main()
{__global__ void VecAdd(float* A, float* B, float* C,int N){ ... }
...
int N = ...;
int size = N * sizeof(float);

float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
float* h_C = (float*) malloc(size);
Initialize(h_A, h_B, N);

float* d_A; float* d_B;float* d_C;
cudaMalloc((void**)&d_C, size);
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);

cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

VecAdd <<<(ceil((float) N/BLOCKSIZE), BLOCKSIZE>>> (d_A, d_B, d_C, N);

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

cudaFree(d_A);cudaFree(d_B); cudaFree(d_C);}
```



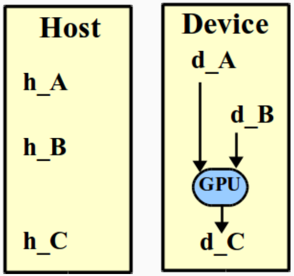
Host-Device Data Transfers. Example

```
int main()
{__global__ void VecAdd(float* A, float* B, float* C,int N){ ... }
...
int N = ...;
int size = N * sizeof(float);

float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
float* h_C = (float*) malloc(size);
Initialize(h_A, h_B, N);

float* d_A; float* d_B;float* d_C;
cudaMalloc((void**)&d_C, size);
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);

cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

VecAdd <<<(ceil((float) N/BLOCKSIZE), BLOCKSIZE>>> (d_A, d_B, d_C, N);

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

cudaFree(d_A);cudaFree(d_B); cudaFree(d_C);}
```



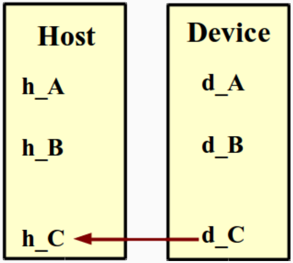
Host-Device Data Transfers. Example

```
int main()
{__global__ void VecAdd(float* A, float* B, float* C,int N){ ... }
...
int N = ...;
int size = N * sizeof(float);

float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
float* h_C = (float*) malloc(size);
Initialize(h_A, h_B, N);

float* d_A; float* d_B;float* d_C;
cudaMalloc((void**)&d_C, size);
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);

cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

VecAdd <<<(ceil((float) N/BLOCKSIZE), BLOCKSIZE>>> (d_A, d_B, d_C, N);

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

cudaFree(d_A);cudaFree(d_B); cudaFree(d_C);}
```



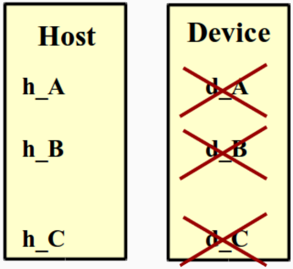
Host-Device Data Transfers. Example

```
int main()
{__global__ void VecAdd(float* A, float* B, float* C,int N){ ... }
...
int N = ...;
int size = N * sizeof(float);

float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
float* h_C = (float*) malloc(size);
Initialize(h_A, h_B, N);

float* d_A; float* d_B;float* d_C;
cudaMalloc((void**)&d_C, size);
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);

cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

VecAdd <<<(ceil((float) N/BLOCKSIZE), BLOCKSIZE>>> (d_A, d_B, d_C, N);

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

cudaFree(d_A);cudaFree(d_B); cudaFree(d_C);}
```



Host-Device Data Transfers. Example

```
int main()
{__global__ void VecAdd(float* A, float* B, float* C,int N){ ... }
...
int N = ...;
int size = N * sizeof(float);

float* h_A = (float*) malloc(size);
float* h_B = (float*) malloc(size);
float* h_C = (float*) malloc(size);
Initialize(h_A, h_B, N);

float* d_A; float* d_B;float* d_C;
cudaMalloc((void**)&d_C, size);
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);

cudaMemcpy (d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy (d_B, h_B, size, cudaMemcpyHostToDevice);

VecAdd <<<(ceil((float) N/BLOCKSIZE), BLOCKSIZE>>> (d_A, d_B, d_C, N);

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

cudaFree(d_A);cudaFree(d_B); cudaFree(d_C);}
```



Host-Device Synchronization

`cudaDeviceSychronize()`

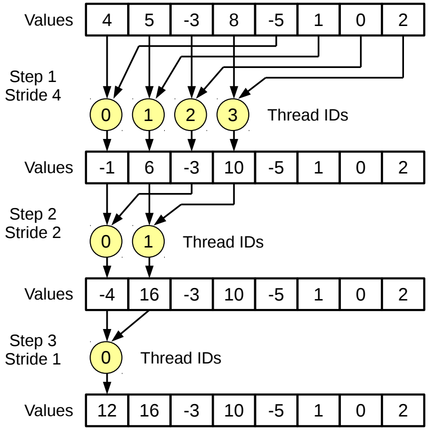
- Used in host code (on the CPU) when it is **desired that CPU activity wait** on the completion of any pending **GPU activity**.
- In many cases, it is not necessary (operations like cudaMemcpy() have an inherent blocking device synchronization).
- For some other purposes **it may be convenient**: debugging code, time measurement, ...
- A kernel launch is asynchronous. It returns control to the CPU thread immediately after launching the kernel (before the kernel has finished executing).

Index

Parallel Reduction Algorithm in GPU

A single value must be obtained from an array of values using an associative binary operator (sum, maximum, minimum, etc.)

1) Each thread of a block loads its associated value of the vector into a shared memory vector.
2) Then, $log_2 B$ steps are performed to obtain the partial result for each block, (B is the size of the thread block). All the threads of a block should synchronize after each step.



CUDA reduction Kernel

```
__global__ void reduceSum(float *d_V, int N)
{
    // Shared memory vector to store the data
    extern __shared__ float sdata[];
    // Compute global index i to access the vector d_V
    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    // Load data into shared memory
    sdata[tid] = ((i < N) ? d_V[i] : 0.0f);
    __syncthreads();

    // Do reduction in shared memory
    for (int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // Write result for this block to global memory
    if (tid == 0) d_V[blockIdx.x] = sdata[0];
}
```

The `extern` keyword in the declaration of the `sdata` means that shared memory will be dynamically allocated for this kernel.

Reduction in a CUDA C Program

```
int main()
{
    ...
    // Kernel launch code
    dim3 threadsPerBlock(128, 1);
    dim3 numBlocks( ceil ((float)(N)/threadsPerBlock.x), 1);
    int smemSize = threadsPerBlock.x*sizeof(float);
    reduceSum<<<numBlocks, threadsPerBlock, smemSize>>>(d_V, N);

    // Perform final reduction in CPU
    cudaMemcpy(h_V, d_V, numBlocks.x*sizeof(float),cudaMemcpyDeviceToHost);
    float sum = 0.0f;
    for (int i=0; i<numBlocks.x; i++)
        sum += h_V[i];
}
```

Notation `<<<numBlocks, threadsPerBlock, smemSize>>>` in the kernel call involves that the `smemSize` bytes of shared memory must be allocated for this kernel.

Reduction algorithms in CUDA

We have presented an unoptimized CUDA reduction kernel (but didactic). Some optimizations can be applied :

- Divide by two the number of thread blocks and replace the single load into shared memory with two memory reads before the first add, in oder to **avoid that half of the threads in each block are idle on the first loop iteration**.
- Perform **loop unrolling**.
- **Process several elements per thread**.

For additional information, refer to the free samples of the CUDA Toolkit (Advanced codes about reduction kernel).

Index

Floyd-Warshall algorithm

- It finds the lengths of the shortest paths between every pair of vertices in a graph $G =< V, E, L >$, where $V = \{v_i, i = 0, \dots, N-1\}$ is a set of N vertices, $E \subseteq V \times V$ is a set of edges connecting the vertices and $L : E \rightarrow \mathbb{R}$ assigns the direct path length to each edge.
- **Input**: adjacency matrix $N \times N$ A for G, where $A_{i,j} = L(v_i, v_j)$ when $(v_i, v_j) \in E$ and a very big constant in other case.
- **Output**: matrix $N \times N$ A in which $A_{i,j}$ contains the shortest path length between v_i to v_j.

```
for k=0 to N-1
    for i=0 to N-1
       for j=0 to N-1
           Aij=min{Aij, Aik + Akj}
```

CUDA kernel to perform the k-th iteration

The Floyd_iter kernel is used to perform the k-th iteration step of the algorithm and each thread updates one different element of the matrix A stored in device memory (d_A).

```
lobal__ void floyd_kernel(int * d_A, const int nverts,
                                              const int k)


 int ij=threadIdx.x+blockDim.x*blockIdx.x;
 if (ij<nverts*nverts)
 { int Aij=d_A[ij];
   int i=ij/nverts;
   int j=ij-i*nverts;
   if (i!=j && i!=k && j!=k)
   { int Aikj=d_A[i*nverts+k]+d_A[k*nverts+j];
     Aij=(Aij>Aikj)? Aikj : Aij;
     d_A[ij]=Aij;}
}
```

Each thread initially computes the position of the element in matrix d_A associated to the current thread.

CUDA Implementation of the Algorithm

```
int main (int argc, char *argv[])
{
    ...
    const int nverts2 = nverts*nverts;
    int *c_Out_A = new int[nverts2]; int *c_In_A = new int[nverts2];
    // Initialize adjacency matrix in c_In_A
    ...
    int size = nverts2*sizeof(int);
    // Allocate device memory for the adjacency matrix
    int *d_A; // Adjacency matrix in device
    cudaMalloc((void **) &d_In_A, size);

    // Transfer host input data to device memory
    cudaMemcpy(d_A, c_In_A, size, cudaMemcpyHostToDevice);
    ...
    // Compute the kernel execution configuration parameters
    int threadsPerBlock = blocksize;
     int blocksPerGrid =(nverts2 + threadsPerBlock - 1) /threadsPerBlock;
    // Main loop with the k update steps
    for(int k=0; k<niters; k++)
       floyd_iter<<<blocksPerGrid,ThreadsPerBlock>>>(d_A, nverts, k);

    // Transfer device output data to host memory
    cudaMemcpy(c_Out_A, d_A, size, cudaMemcpyDeviceToHost);
    ...
}
```

Index

1D Linear Advection

1D Linear Advection Equation (1DLAD)

$$\frac{\partial \phi}{\partial t} + u \frac{\partial \phi}{\partial x} = 0,$$

- $\phi = \phi(x, t)$ is a real function with $x \in [a, b]$ and $t > 0$.
- Scalar $u \in \mathbb{R}$: propagation speed along x-direction (constant).
- Initial state: $\phi(x, 0) = \phi_0(x)$.
- We assume homogeneous Neumann boundary conditions

$$\frac{\partial \phi}{\partial x}(a, t) = \frac{\partial \phi}{\partial x}(b, t) = 0.$$

- Exact solution: $\phi(x, t) = \phi_0(x - ut)$.

The Lax-Friedrichs method

- **Discretization** using uniform grids:
  - $n + 1$ Spatial grid points:

$$x_i, \quad i = 0, \ldots, n, \text{ where } x_i = a + i\Delta x, \qquad \text{with } \Delta x = \frac{b-a}{n}.$$

  - M Time steps: $t^1, t^2, \ldots, t^M, \quad (t^{k+1} = t^k + \Delta t).$

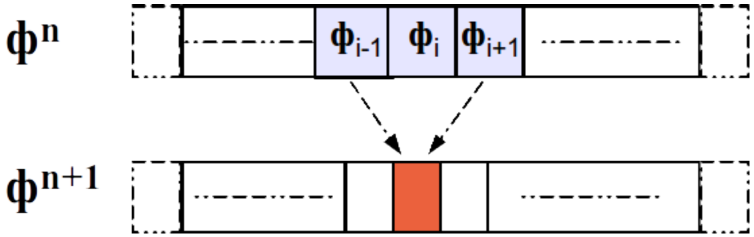
Lax-Friedrichs FD scheme to solve 1DLAD Eq.

$$\phi_i^{k+1} = \frac{1}{2}\left(\phi_{i-1}^k + \phi_{i+1}^k - \frac{u\,\Delta t}{\Delta x}(\phi_{i-1}^k - \phi_{i+1}^k)\right),$$

$\phi_i^k = \phi(x_i, t^k), \quad i = 0, \ldots, n, \quad k = 1, \ldots, M.$

- **Boundary conditions**: $\phi_{-1}^k = \phi_0^k, \qquad \phi_{n+1}^k = \phi_n^k.$

2-point stencil of the Lax-Friedrichs method



A sequential C program to implement the method

- At each step, each data element of `phi_new` is computed from two neighbouring elements of `phi`.
- `phi` vector includes elements at the edges to store the ghost values.

```
void swap_pointers(float * *a, float * *b)
  {float * tmp=*a;*a=*b;*b=tmp;}
...
int main(int argc, char* argv[]){
   // Define Delta x, Delta t, velocity (u) and the Courant number
  int n= ... ;float dx = ...;float dt=..; float u=...; float cu = u*dt/dx;
  //Declare the initial and final state vectors
  float * phi   =new float[n+3]; float * phi_new=new float[n+3];
  ..
  Initialize(phi); // Set phi values at t=0

  for(int k=1; k <= M; k++)  // Time steps
   {// Impose Homogeneous Neumann Boundary Conditions
    phi[index(-1)] =phi[index(0)]; phi[index(n+1)]=phi[index(n)];

    for(int i=0;i<=n;i++) //Lax-Friedrichs Update
         phi_new[index(i)]=0.5*((phi[index(i+1)]+phi[index(i-1)])
                          -cu*(phi[index(i+1)]-phi[index(i-1)]));

    swap_pointers (&phi,&phi_new);
   }
```

A CUDA Linear Advection solver (1)

- Calculation of all elements of `phi_new` for a time step can be performed in parallel.
- **i-th Thread could compute `phi_new[i]`.**
- We declare a kernel function with 4 arguments:
  - A pointer to a vector denoting the initial state vector `d_phi` (ϕ^k).
  - A pointer to the new state vector `d_phi_new` (ϕ^{k+1})
  - An integer `cu` (`cu` $= \frac{u\Delta t}{\Delta x}$)
  - An integer `n` denoting the vector size minus 2.

```
global__ void FD_kernel1 (float * d_phi,
                          float * d_phi_new,
                          float cu, const int n)
```

A CUDA Linear Advection solver. Main program

```
#define BLOCKSIZE 256

... //FD_kernel function definition

int main(int argc, char* argv[]){
  int size=(n+3)*sizeof(float);
  float * d_phi=NULL; float * d_phi_new=NULL;
  cudaMalloc((void **) &d_phi, size); cudaMalloc((void **) &d_phi_new, size);

  Initialize(phi); // Set phi values at t=0

  cudaMemcpy(d_phi,phi,size,cudaMemcpyHostToDevice);

  for(int k=1; k <= M ;k++)    // Time Steps
   {
    int blocksPerGrid =(int) ceil((float)(n+1)/BLOCKSIZE);

    // ********* CUDA Kernel Launch ***********************************
    FD_kernel1<<<blocksPerGrid, BLOCKSIZE >>> (d_phi, d_phi_new, cu, n);
    swap_pointers (&d_phi,&d_phi_new);
   }

  cudaMemcpy(phi_GPU, d_phi, size, cudaMemcpyDeviceToHost);
}
```

A CUDA Linear Advection solver. Kernel def.

- Each thread reads two elements of the same input vector except for two threads which read an additional element to impose the boundary conditions.
- The work of threads which are not assigned to elements of `d_phi_new` is avoided.

```
__global__ void FD_kernel1(float * d_phi,
                             float * d_phi_new,
                                  float cu, int n)

 int i=threadIdx.x+blockDim.x*blockIdx.x+1;

 // Lax-Friedichs Stencil
 if (i<n+2)
   d_phi_new[i]=0.5*((d_phi[i+1]+d_phi[i-1])
                          -cu*(d_phi[i+1]-d_phi[i-1]));

 // Impose Boundary Conditions
 if (i==1)   d_phi_new[0]=d_phi_new[1];
 if (i==n+1) d_phi_new[n+2]=d_phi_new[n+1];
```

Tiling algorithms

- **Previous kernel**: The number of floating-point calculations per global memory access is low. This prevents us from achieving high performance.
- **Approach to improve**: it involves to replace global memory accesses with shared memory accesses.
- It can involve to reorganize the code to reuse data in shared memory.
- **Tiling approach** consists of two steps:
  1. Threads of a block collaborate to load a small piece of the input data elements (*tile*), into shared memory before using them.
  2. Threads use several times elements of the tile by accessing the shared memory.
- **Size of the tiles**: must fit into the shared memory.

A tiled CUDA Advection solver

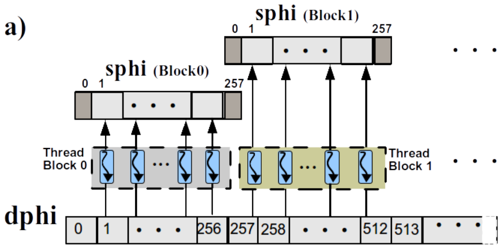
- All elements of `d_phi`, needed to compute all elements of `d_phi_new` for a block are loaded into shared memory.
- The number of elements to be loaded must be equal to `BLOCKSIZE + 2` (two additional elements at the edges).
- A shared memory array, `s_phi`, is declared to hold elements needed to compute `BLOCKSIZE` elements of `d_phi_new`.
- Tile is loaded in shared memory in two phases.
- After loading the tile, `__syncthreads()` function is invoked to ensure all threads in the block have finalized the load.
- When the tile is loaded for a block, we can compute the `BLOCKSIZE` elements of `d_phi_new` reading only from `s_phi`.

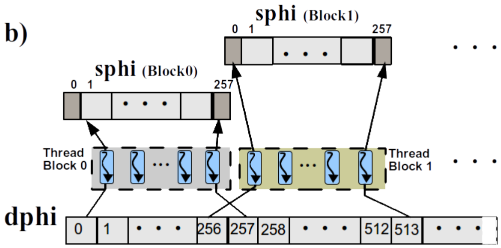
Loading the tile in shared memory

Tile is loaded in shared memory in two phases.

a) **Load internal elements**

b) **Load the halo elements**: The first and last threads in the block loads the elements at the edges.





A tiled CUDA Advection kernel (1)

```
__global__ void FD_kernel2 (float * d_phi, float * d_phi_new,
                                              float cu, const int n)
 {
   int li=threadIdx.x+1; //local index in shared memory vector
   int gi=  blockDim.x*blockIdx.x+threadIdx.x+1; // global memory index
   int lstart=0; // start index in the block (left value)
   int lend=BLOCKSIZE+1; // end index in the block (right value)

   __shared__ float s_phi[BLOCKSIZE + 2]; //shared mem. vector
   ...
   // a) Load internal points of the tile in shared memory
   if (gi<n+2) s_phi[li] = d_phi[gi];

   // b) Load the halo points of the tile in shared memory
   if (threadIdx.x == 0) // First Thread (in the current block)
      s_phi[lstart]=d_phi[gi-1];

   if (threadIdx.x == BLOCKSIZE-1) // Last Thread
         if (gi>=n+1) // Last Block
            s_phi[(n+2)%BLOCKSIZE]=d_phi[n+2];
         else
            s_phi[lend]=d_phi[gi+1];

   __syncthreads(); // Barrier Synchronization

   ...
```

A tiled CUDA Advection kernel (2)

```
__global__ void FD_kernel2 (float * d_phi, float * d_phi_new,
                                             float cu, const int n)
 {
   ...
   __shared__ float s_phi[BLOCKSIZE + 2]; //shared mem. vector
   float result;

       ... Load Tile (previous slide)

   __syncthreads();

  if (gi<n+2)
   {
    // Lax-Friedrichs Update
     result=0.5*((s_phi[li+1]+s_phi[li-1])
         -cu*(s_phi[li+1]-s_phi[li-1]));
     d_phi_new[gi]=result;
   }

  // Impose Boundary Conditions
  if (gi==1) d_phi_new[0]=d_phi_new[1];
  if (gi==n+1) d_phi_new[n+2]=d_phi_new[n+1];

 }
```

Programming Resources

- John Cheng, Max Grossman, Ty McKercher (2014) **Professional CUDA C Programmingh**, Morgan Kaufmann.
- Kirk D., Hwu Wen-mei (2012) **Programming Massively Parallel Processors, Second Edition: A Hands-on Approach**, Morgan Kaufmann.
- Sanders J., Kandrot E (2010) **CUDA by Example: An Introduction to General-Purpose GPU Programming**, Addison Wesley.
- **CUDA C Programming Guide**. NVIDIA. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.
- **CUDA Samples**: Code samples that are included with the NVIDIA CUDA Toolkit `http://docs.nvidia.com/cuda/cuda-samples/`