Introduction to CUDA programming for physicists

François Gelis

IPhT, Saclay

Institut de
Physique Théorique
CEA/DRF Saclay

Lecture I

Outline

- Architecture of GPUs
- What tasks are GPUs good at?
- Overview of CUDA

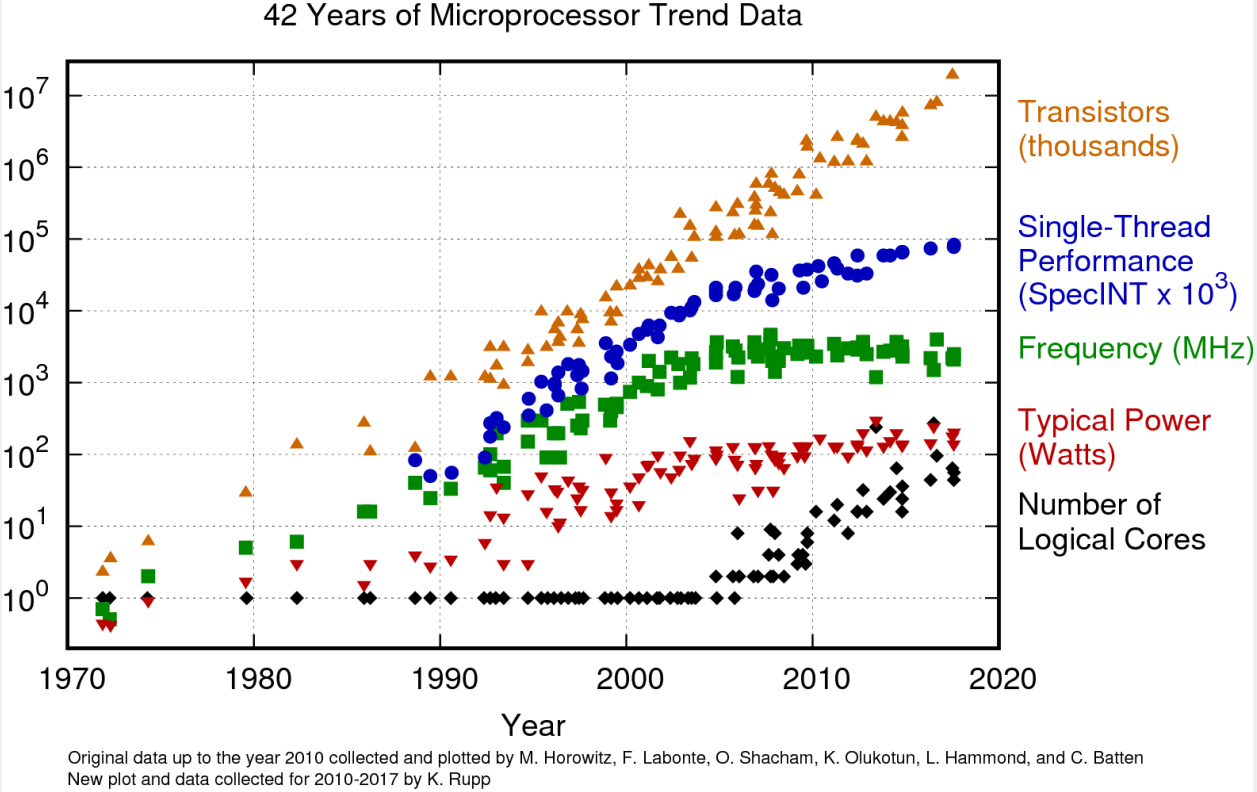
- Diagnosing errors
- Memory management (allocation, copy)
- CUDA kernels

- Streams, Synchronization
- Shared memory
- Reduction operations

- Memory performance tuning
- How many threads?
- CUDA libraries for common tasks: cuFFT, cuBLAS, ...

Introduction

CPU IMPROVEMENTS IN THE PAST 40 YEARS



- Single thread performance reaches a plateau, correlated to the plateau in frequency, which comes from constraints of thermal dissipation
- Performance increases mostly by increasing the number of cores

Typical CPUs

- Can have 2 (low end) to 128 (very high end) cores
- These cores share main memory, and some of the cache
- The threads running on different cores can perform totally unrelated tasks
- The operating system takes care of sharing the available cores among the running threads (by giving each of them time-slices to use a core)
- Parallel programming can be done with OpenMP

What is a GPU?

- GPU = Graphics Processing Unit. Originally, in charge of the intensive parts of graphics manipulations
- Each pixel in an image is independent of the others $\Longrightarrow$ highly parallelizable task. GPUs can have from a few hundred cores (low end) to ~ 20000 cores (high end)
- High accuracy not necessary for graphics $\Longrightarrow$ optimized for single precision, with some shortcuts in the accuracy of certain functions (but recent generations of GPUs are fully compliant with common standards for floating point accuracy)
- Most GPUs have higher performance in single precision than double precision (sometimes even higher for reduced formats, like 16-bit floating point)
- Perform much better if all threads execute the same instruction (Single Instruction Multiple Data = SIMD)

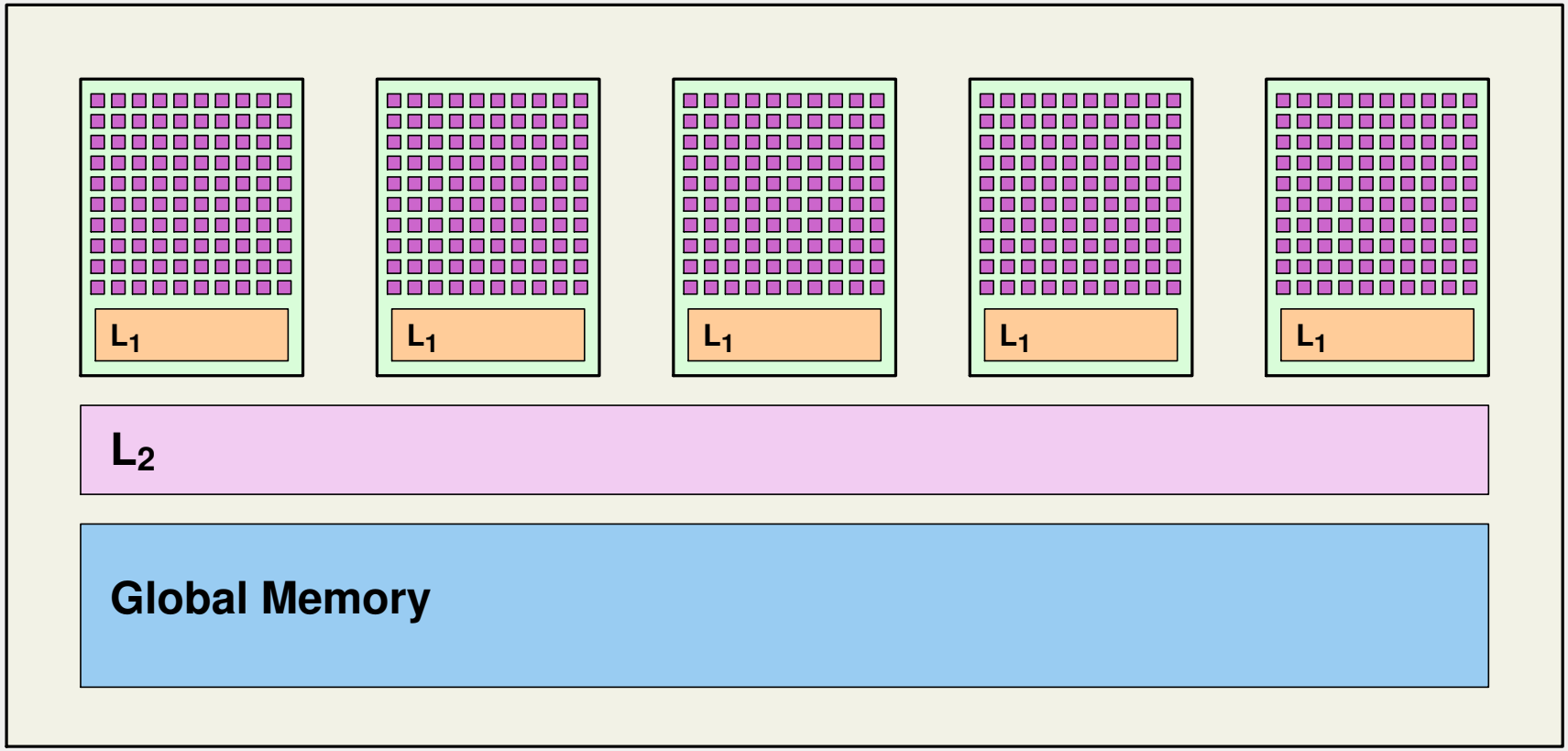
Where can one do computations on GPUs?

- At IPhT:
  - some personal computers/laptops have GPUs
  - akira.ipht.fr: 2 x NVIDIA TITAN X

- National computing resources administered by GENCI:
  - TGCC: V100
  - IDRIS: V100, A100, soon: H100
    (light application procedure for requests up to 50000 GPU.hour)
  - CINES: L40

Architecture of GPUs

What's inside a GPU?



What's inside a GPU?

- Computing cores (many: 100's to 10,000's)
- Memory: from 1-2 GB to 192 GB
  - L1 cache: on the same chip as the cores, fast, shared by all threads in a block
  - L2 cache: off-chip, slower (latency ~ 100 clock cycles)
  - Global memory: off-chip, even slower (latency ~ 500 clock cycles), but much larger capacity
- Fast link to the host computer (32 to 128 GB/s, in each direction)
- Fast direct GPU-to-GPU links (for multi-GPU systems)

- Notes:
  - A GPU devotes more hardware to computing and less hardware to flow control. Designed primarily for executing common instructions in all threads
  - Part of the L1 cache can be used as "shared memory", i.e., very fast user controllable memory seen by all threads in a block

Examples of GPUs: NVIDIA Quadro T1000 (in my laptop)

- 896 cores
- 4GB memory
- Peak performance:
  - single precision: 2.6 TeraFLOPS
  - double precision: 81 GigaFLOPS (1/32)

Examples of GPUs: NVIDIA V100

- 5120 cores
- 32GB memory
- Peak performance:
  - single precision: 15.6 TeraFLOPS
  - double precision: 7.8 TeraFLOPS (1/2)

Examples of GPUs: NVIDIA A100

- 6912 cores
- 80GB memory
- Peak performance:
  - single precision: 19.5 TeraFLOPS
  - double precision: 9.7 TeraFLOPS (1/2)

Examples of GPUs: NVIDIA H100

- 16896 cores
- 80GB memory
- Peak performance:
  - single precision: 66.9 TeraFLOPS
  - double precision: 33.5 TeraFLOPS (1/2)

Performance per Watt

- A very important concern in the design of a large computer center is its power requirements

- NVIDIA A100: 6912 cores, 19.5 TFlops, 400W, 49 GFlops/W
- NVIDIA H100: 16896 cores, 66.9 TFlops, 700W, 96 GFlops/W

- AMD 7995WX (CPU): 96 cores, 12.2 TFlops, 350W, 35 GFlops/W

What tasks are GPUs good at?

- Very good at tasks where all threads execute exactly the same sequence of instructions (on different data)
- Best if successive threads access successive addresses in memory
- Best if the ratio of computations to memory accesses is large
- Best if the data stays for a long time on the GPU (frequent transfers to and from the GPU are detrimental to the performance)

- Not so good for code where threads follow different logical branches (“divergent code”)
- Not so good with “random” memory access

Brands of GPUs

- Intel
- AMD
- **NVIDIA** : can be used with CUDA

Listing all NVIDIA GPUs

- The command nvidia-smi lists all the NVIDIA GPUs available on the computer, their main properties and current usage
- Example: on akira.ipht.fr, this command outputs

```
Mon Sep 16 10:30:21 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 545.23.08              Driver Version: 545.23.08    CUDA Version: 12.3     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA TITAN Xp                Off | 00000000:01:00.0 Off |                  N/A |
| 72%   87C    P2             178W / 250W |   2034MiB / 12288MiB |    100%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA TITAN Xp                Off | 00000000:02:00.0 Off |                  N/A |
| 64%   86C    P2             185W / 250W |   5796MiB / 12288MiB |    100%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A     17359      C   python3                                   2032MiB |
|    1   N/A  N/A     27085      C   python3                                   3010MiB |
+---------------------------------------------------------------------------------------+
```

Goals of this course

- **Goal** 1: write correct CUDA programs
  - compile without errors
  - run without errors
  - produce the expected output

- **Goal** 2: write fast CUDA programs
  - use lots of threads
  - good memory access patterns

Overview of CUDA

Programming frameworks for GPUs

- **OpenACC** (for *open accelerators*): similar to OpenMP, works by adding annotations (#pragma ...) in an existing code to hint the compiler to offload certain computations to a GPU. Works on various brands of GPUs. Offers less control about what is really going on on the GPU.
- **OpenCL**: C-like language to write programs that use GPUs. Code in principle portable across various brands of GPUs, but with varying performances.

- **CUDA**: extensions of C/C++ to run code on NVIDIA GPUs. Not portable to other brands of GPUs, but better optimized on NVIDIA GPUs (thanks to the software model matching closely the features available on hardware).

What is CUDA?

- CUDA source files usually have an extension .cu (but this is not mandatory)
- CUDA follows the syntax of C++
- When mixing CUDA and plain C, the only complication is that C++ functions are by default not callable from C. One should declare them with extern "C", as in

```
extern "C"  function(arguments){
  instruction1;
  instruction2;
  ...
}
```

- There exist wrappers to use CUDA in Fortran, in Python, and in a few other programming languages

Main classes of functions available in CUDA

- All standard functions of the C math library (such as sin, cos, exp, ...), both in single and double precision, for execution on a GPU. (The function names are the same, and the compiler determines automatically which binary instruction to produce from the context)
- Functions to transfer data to and from a GPU (between GPU and HOST, or between two GPUs)
- Functions to start a parallel computation on a GPU
- Functions to control the scheduling between tasks

Main steps in a typical GPU computation

- Allocate space for data on the GPU
- Populate this space (in general, by copying data from the HOST to the GPU)
- Perform one or more computations on this data on the GPU
- Retrieve result from the GPU to the HOST

First example - CPU version

```c
#include <omp.h>
#define REPEAT 256

float f(float x,float y){                          // Dummy function...
  return sinf(cosf(sinf(sqrtf(powf(cosf(x*x+y*y),2.0f)))));
}

int main(void){
  long unsigned int N = 1024L*1024*256;            // Allocate two arrays of size N
  float *a = (float *)malloc(N*sizeof(float));
  float *b = (float *)malloc(N*sizeof(float));

  for (long unsigned int i=0;i<N;i++){             // Fill the arrays with random numbers
    a[i] = (float)rand()/(float)(RAND_MAX);
    b[i] = (float)rand()/(float)(RAND_MAX);
  }

  for (int count=0;count<REPEAT;count++) {         // Process the arrays
#pragma omp parallel for num_threads(6)            // Use OpenMP to parallelize the loop
    for (long unsigned int i=0;i<N;i++) a[i] = f(a[i], b[i]);
  }

  exit(0);
}
```

First example - GPU version

```
#include <cuda.h>
#include <cuda_runtime.h>
#define REPEAT 256

__device__ float f(float x,float y){                          // Function that runs on GPU
  return  sinf(cosf(sinf(sqrtf(powf(cosf(x*x+y*y),2.0f)))));
}

__global__ void apply_function(float * a,float * b){          // CUDA ''kernel''
  int id   = blockIdx.x * blockDim.x + threadIdx.x;
  a[id] = f(a[id],b[id]);
}

int main(void){
  long unsigned int N = 1024L*1024*256;                       // Allocate arrays on HOST
  float *a = (float *)malloc(N*sizeof(float));
  float *b = (float *)malloc(N*sizeof(float));

  for (long unsigned int i=0;i<N;i++){                        // Fill with random numbers on HOST
    a[i] = (float)rand()/(float)(RAND_MAX);
    b[i] = (float)rand()/(float)(RAND_MAX);
  }

  float *d_a,*d_b;
  cudaMalloc(&d_a,N*sizeof(float));                           // Allocate two arrays on GPU
  cudaMalloc(&d_b,N*sizeof(float));
  cudaMemcpy(d_a,a,N*sizeof(float),cudaMemcpyHostToDevice);   // Copy data from HOST to GPU
  cudaMemcpy(d_b,b,N*sizeof(float),cudaMemcpyHostToDevice);

  for (int count=0;count<REPEAT;count++){                     // Process arrays on GPU
    apply_function<<<N/128,128>>>(d_a,d_b);
  }
  cudaDeviceSynchronize();

  cudaMemcpy(a,d_a,N*sizeof(float),cudaMemcpyDeviceToHost);   // Copy result to HOST

  exit(0);
}
```

First example - GPU version: headers

```
#include <cuda.h>
#include <cuda_runtime.h>
```

- In C or C++, these headers provide the definitions of all CUDA related functions. They need to be included at the top of all files that use CUDA functions

First example - GPU version: function decorations

```
__device__ float f(float x,float y){
  return  sinf(cosf(sinf(sqrtf(powf(cosf(x*x+y*y),2.0f)))));
}

__global__ void apply_function(float * a,float * b){
  int id   = blockIdx.x * blockDim.x + threadIdx.x;
  a[id] = f(a[id],b[id]);
}
```

- `__device__` before a function declaration tells the compiler to produce a function that will run on the GPU
- `__host__` produces a function for the CPU (this is the default and it can be omitted)
- `__device__ __host__` produces a function that runs on GPU and CPU (it actually produces two versions of the function)
- `__global__` indicates a “CUDA kernel”, i.e., a function that applies the same sequences of instructions to all elements of an array, and runs on the GPU

First example - GPU version: memory operations

```
cudaMalloc(&d_a,N*sizeof(float));
cudaMemcpy(d_a,a,N*sizeof(float),cudaMemcpyHostToDevice);
```

- `cudaMalloc()`: allocates memory on the GPU. Arguments:
  - Address of a pointer (when done, the pointer contains the memory address on GPU of the allocation)
  - Size of the allocation
- `cudaMemcpy()`: copy memory to and from the GPU. Arguments:
  - Pointer to the destination
  - Pointer to the source
  - Size of the chunk to be copied
  - `cudaMemcpyHostToDevice`: direction of the copy (also: `cudaMemcpyHostToHost`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, `cudaMemcpyDefault`)

First example - GPU version: CUDA kernel call

```
apply_function<<<N/128,128>>>(d_a,d_b);
```

- *name*`<<<# blocks,# threads/block>>>(arguments)`: launches a CUDA kernel with
  (# threads) = (# blocks) $\times$ (# threads/block)
  - Returns to the host before completion
  - By default, successive kernels are executed sequentially
  - Scalar arguments are copied on the fly to the GPU
  - Arrays must already be present on the GPU
  - The block # is `blockIdx.x`, and the thread # is `threadIdx.x`. The block size is `blockDim.x`
  - `blockIdx.x * blockDim.x + threadIdx.x` is the absolute thread index (can be used as index into an array)
    Note: threads and blocks are arranged in a grid that can have up to 3 dimensions (replace `.x` by `.y` or `.z`)
  - All threads in a block run on the same chip and see the same L1 cache (or the same shared memory)

First example - GPU version: synchronization

```
cudaDeviceSynchronize();
```

- `cudaDeviceSunchronize()`:
  - Called by the host
  - Blocks until all queued CUDA calls have completed
  - Can be called to safely use data on GPU
  - With multiple GPUs, applies only to the currently active GPU
  - Note: there are more fine grained ways of waiting for the completion of a subset of CUDA calls

First example - CUDA Compilation

```
nvcc -O3 --use_fast_math -o test example-0-gpu.cu -lcuda -lcudart -lstdc++
```

- `nvcc`: NVIDIA's CUDA compiler (`nvcc` compiles the CUDA instructions, and passes the rest to the host compiler: gcc, Intel's compiler, ...)
- `-O3 --use_fast_math`: optimization flags
- `-o test`: specify the name of the executable
- `-lcuda -lcudart -lstdc++`: linked libraries (contain the actual machine code for all functions used in the program)

- Note: `nvcc` knows where to find all CUDA header files and libraries, so there is no need to specify their location

First example - Timings

- Timing on CPU (with 6 cores):

```
Processing arrays : 63.27 seconds
```

- Timing on GPU (Quadro T1000):

```
Processing arrays
        Allocation       : 0.08
        Copy HOST -> GPU : 0.36
        Processing       : 4.61
        Copy GPU -> HOST : 0.17
Total : 5.22 seconds
```

- Notes:
  - GPU 12 times faster than CPU on this example
  - Overheads (allocations, transfers) not totally negligible
  - GPU beneficial only if there are enough computations to compensate these overheads

First example - Variant

```
__global__ void apply_function_REPEAT(float * a,float * b){  // CUDA ''kernel''
  int id   = blockIdx.x * blockDim.x + threadIdx.x;
  int count;
  for (count=0;count<REPEAT;count++) a[id] = f(a[id],b[id]); // loop now inside kernel
}

int main(void){
  [...]
  apply_function_REPEAT<<<N/128,128>>>(d_a,d_b);            // process arrays on GPU
  [...]
}
```

- The loop from 1 to REPEAT=256 is now included directly inside the CUDA kernel
- Only one kernel call instead of 256
- Processing time goes from 4.6" to 1.4"
- Each kernel call has an overhead $\sim$ 10ms
- Whenever possible: reduce the number of kernel calls by having kernels that do more work

First example - Using 2 GPUs

```
[...]
float *d_a1,*d_b1,*d_a2,*d_b2;
cudaSetDevice(0);
cudaMalloc(&d_a1,(N/2)*sizeof(float));
cudaMalloc(&d_b1,(N/2)*sizeof(float));
cudaSetDevice(1);
cudaMalloc(&d_a2,(N/2)*sizeof(float));
cudaMalloc(&d_b2,(N/2)*sizeof(float));

cudaSetDevice(0);
cudaMemcpy(d_a1,a,(N/2)*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_b1,b,(N/2)*sizeof(float),cudaMemcpyHostToDevice);
cudaSetDevice(1);
cudaMemcpy(d_a2,a+N/2,(N/2)*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_b2,b+N/2,(N/2)*sizeof(float),cudaMemcpyHostToDevice);

cudaSetDevice(0);
apply_function_REPEAT<<<(N/2)/128,128>>>(d_a1,d_b1);
cudaSetDevice(1);
apply_function_REPEAT<<<(N/2)/128,128>>>(d_a2,d_b2);

cudaSetDevice(0);
cudaMemcpy(a,d_a1,(N/2)*sizeof(float),cudaMemcpyDeviceToHost);
cudaSetDevice(1);
cudaMemcpy(a+N/2,d_a2,(N/2)*sizeof(float),cudaMemcpyDeviceToHost);
[...]
```

First example - Using 2 GPUs

- Use cudaSetDevice(n) to set which GPU is active for subsequent commands (memory copies, kernel launches,...)
- Note: the same kernels can be used for all GPUs (but the compiler would have to be told if there are several types of GPUs that require distinct binary code)
- The argument n ranges from 0 to $\mathrm{N_{GPU}} - 1$
- The mapping from GPUs to the index n is defined at boot time (not user controllable). It is often not necessary to know it

- Potential for improvement: cudaMemcpy() is *synchronous*, i.e., it returns control to the host only when the transfer is complete, so transfers to GPU-0 and GPU-1 do not happen in parallel

First example - Timings using 2 GPUs

- Tested on a node of jean-zay.idris.fr (with 2×V100 GPUs)
- Timing on CPU (with 20 cores):

```
Processing arrays : 25.48 seconds
```

- Timing on 1 GPU:

```
Processing arrays
        Allocation       : 0.19
        Copy HOST -> GPU : 0.46
        Processing       : 1.09
        Copy GPU -> HOST : 0.23
Total : 1.96 seconds
```

- Timing on 1 GPU (with loop inside kernel):

```
Processing arrays
        Allocation       : 0.18
        Copy HOST -> GPU : 0.46
        Processing       : 0.30
        Copy GPU -> HOST : 0.23
Total : 1.17 seconds
```

- Timing on 2 GPUs (with loop inside kernel):

```
Processing arrays
        Allocation       : 0.28
        Copy HOST -> GPU : 0.45
        Processing       : 0.15
        Copy GPU -> HOST : 0.22
Total : 1.11 seconds
```

CUDA RESOURCES

- https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

  CUDA online reference manual. Detailed description of all available functions, grouped by topics. Very useful if one needs a description of a function's behavior and syntax

- Blog entries on NVIDIA website. Example: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

  Quick discussion of a given topic, accompanied with working examples

- Google a CUDA function name...

Lecture II

Outline

- Architecture of GPUs
- What tasks are GPUs good at?
- Overview of CUDA

- Diagnosing errors
- Memory management (allocation, copy)
- CUDA kernels

- Streams, Synchronization
- Shared memory
- Reduction operations

- Memory performance tuning
- How many threads?
- CUDA libraries for common tasks: cuFFT, cuBLAS, ...

Diagnosing errors

My CUDA program is not working. What shall I do?

- Not an infrequent situation: a CUDA program crashes, or runs but produces garbage
- Debugging highly parallel code is delicate (because the error could affect a single thread out of many)
- Common causes of problems:
  - Using an uninitialized pointer
  - Using a host/device pointer in the wrong context
  - Reading/writing beyond the size of a given allocation
  - Bad scheduling among dependent tasks
- Tip: Use a different naming convention for host and device pointers, e.g., `h_...` for host pointers and `d_...` for device pointers

The "print method"

- One can use standard print statements within CUDA code:

```
printf("a=%.6f b=%.6f\n",variable1,variable2);
```

  in order to inspect the content of some variables

- Caveats:
  - If several threads print out something, there is no guarantee that the outputs are ordered by thread index
  - If used without caution inside a kernel called with thousands of threads, the output will be overwhelming (and useless)
    One should usually put a condition such that only one (or a few) thread prints something, e.g.,

```
if (blockIdx.x * blockDim.x + threadIdx.x == 0) {
  printf("a=%.6f b=%.6f\n",variable1,variable2);
 }
```

Finding "NaN" 's

- Reading beyond the boundary of an array often reads garbage (i.e., random bits that are not a valid representation for the expected type), which quickly results in "NaN" (Not a Number) appearing in the data one manipulates
- CUDA has a function isnan() that takes a float or double and returns 0 if it is not a NaN and 1 if it is a NaN
- The same function exists also in the standard C library for checking variables that live on the host
- The test reads

```
if (isnan(variable)) {
  do_something;
 }
```

- On can check in parallel for NaN's in a large array on the GPU (see the section on reduction operations in next course)

Finding "NaN" 's

- Note: when compiling with optimization flags set, some compilers remove the test. One can avoid this by using:

```
int isnan_d(double d){
  uint64_t u;
  memcpy(&u,&d,8);
  return ((u&0x7ff0000000000000ULL) == 0x7ff0000000000000ULL)&&(u&0xfffffffffffffULL);
}


int isnan_f(float f) {
    uint32_t u;
    memcpy(&u,&f,4);
    return ((u&0x7f800000) == 0x7f800000)&&(u&0x7fffff);
}
```

(the seemingly magic numbers in these functions will catch the specific bit representation of a float or double NaN, respectively)

Finding faulty CUDA calls

- Some CUDA calls return before the completion of their task, and failures may appear significantly after the command was issued
- In order to find which command caused the problem, one may check for errors after CUDA calls. The following code must be inserted at the top of the CUDA source file:

```
#define cudaCheckErrors(msg) __getLastCudaError(msg, __FILE__, __LINE__)

inline void __getLastCudaError(const char *errorMessage, const char *file,
                               const int line) {
  cudaError_t err = cudaGetLastError();

  if (cudaSuccess != err) {
    fprintf(stderr,
            "%s(%i) : CUDA error :"
            " %s : (%d) %s.\n",
            file, line, errorMessage, static_cast<int>(err),
            cudaGetErrorString(err));
    exit(EXIT_FAILURE);
  }
}
```

(or put in a separate file error.cu , and then included with #include "error.cu")

Finding faulty CUDA calls

- A sequence of CUDA calls

```
cuda_call_1;
cuda_call_2;
cuda_call_3;
```

  can be modified to check for errors after each call:

```
cuda_call_1;
cudaCheckErrors("call 1");
cuda_call_2;
cudaCheckErrors("call 2");
cuda_call_3;
cudaCheckErrors("call 3");
```

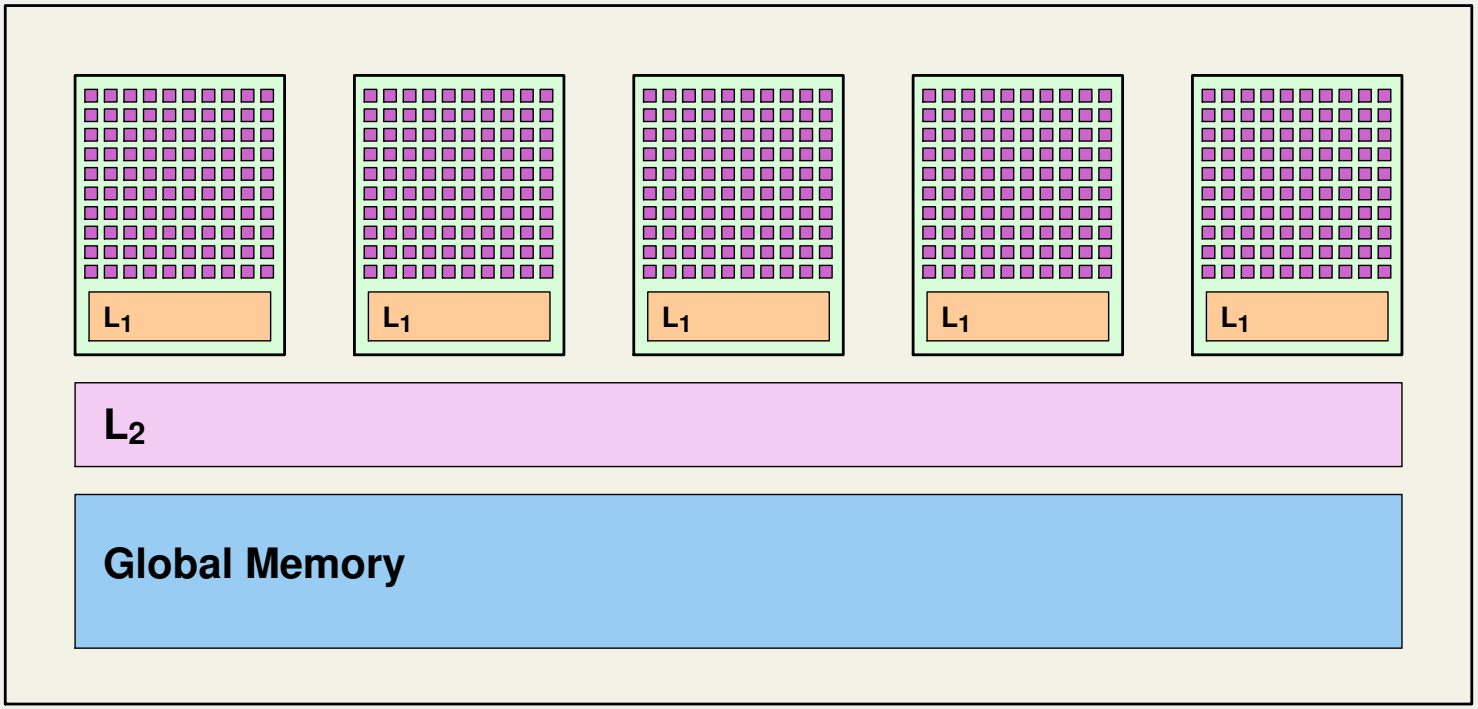
- The tags used in cudaCheckErrors("tag") can be chosen freely. But, to be useful, they should be unique in order to identify where the error occurred
- If an error is detected, the program exits immediately and prints some info about the error (tag, source file, line number). But these error messages are often a bit cryptic..

Memory management

Overview

- CUDA provides a set of functions to allocate, free, set or copy chunks of data in the global memory of a GPU
- The list of these functions, with their detailed description, can be found at https://docs.nvidia.com/cuda/cuda-runtime-api/group_CUDART_MEMORY.html

Memory layout of a GPU



- Both the global memory and the L1 cache (shared memory) are accessible through programming. The L2 cache is handled automatically in hardware
- In this section, we discuss only global memory. See later to see why and how to use L1 cache as shared memory

cudaMalloc()

- Syntax:

```
float *p;
cudaMalloc(&p,N*sizeof(float));
```

- CAUTION: The allocated memory region is not cleared (it may contain garbage, or data from a previous allocation)
- The start address is properly aligned for all types
- The size (second argument) is expressed in bytes
- cudaMalloc returns an error in case of failure. It is advised to check this (I never do it...)

cudaFree()

- Syntax:

```
float *p;
cudaMalloc(&p,N*sizeof(float));
[...]
cudaFree(p);
```

- This function can only free memory that has been allocated with cudaMalloc (or one of its variants), not host memory allocated with the host malloc
- CAUTION: Calling cudaFree multiple times with the same pointer produces an error

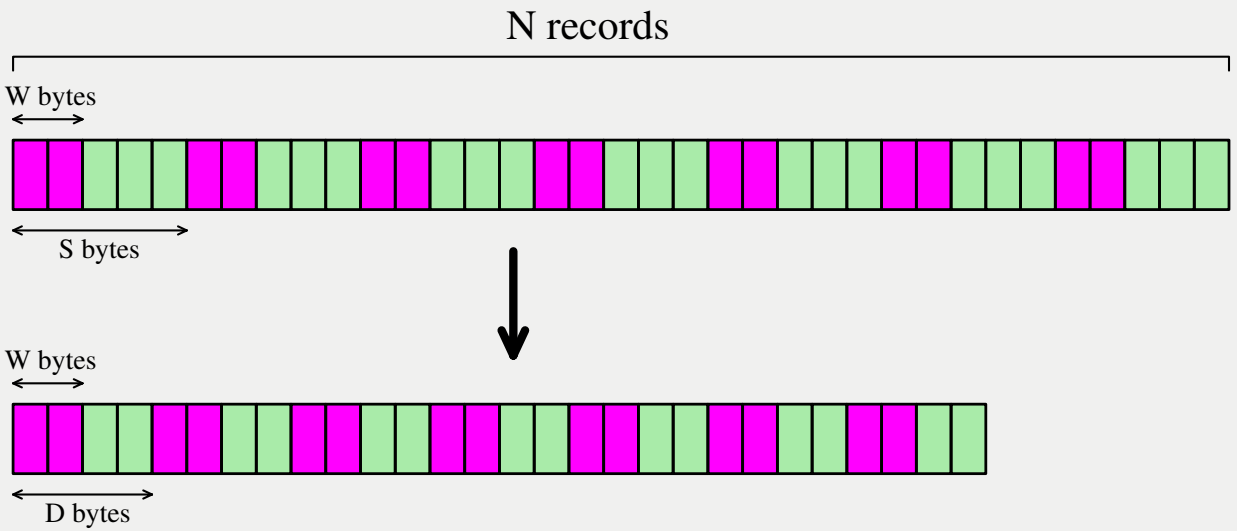
cudaMemcpy()

- Syntax:

```
cudaMemcpy(destination,source,size,type_of_transfer);
```

- The source and destination pointers must correspond to sufficiently large memory allocations (at least as large as size)
- The size is usually specified as N*sizeof(type)
- The type_of_transfer is one of:
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice (the two devices are the same)
  - cudaMemcpyDefault (the type is guessed from pointer values)
- Synchronous with respect to the host (returns only after the copy is completed)

cudaMemcpy2D()



- Syntax:

```
cudaMemcpy2D(destination,D,source,S,W,N,type_of_transfer);
```

- Allows to copy strided data with a single command
- The source and destination arrays must be large enough to contain the data being copied and the gaps in between

cudaMemset()

- Syntax:

```
cudaMemset(address,value,size);
```

- Sets size bytes to value, starting at address
- Often used with value=0 to initialize to zero a memory region
- CAUTION: value is truncated to unsigned char (i.e., one byte) before memory is set. Use the function cuMemsetD32(address,value,N) to set the array that starts at address to the 4-byte value, repeated N times

Asynchronous functions

- All the functions described so far are synchronous. They return only after having completed their task
- There exists an *asynchronous* version of these functions, that return to the host before completion:
  - cudaMallocAsync(...,stream)
  - cudaFreeAsync(...,stream)
  - cudaMemcpyAsync(...,stream)
  - cudaMemcpy2DAsync(...,stream)
  - cudaMemsetAsync(...,stream)
- The extra argument stream is of type *cudaStream_t* (the default stream, 0, can be omitted), and is created by the function cudaStreamCreate

Asynchronous functions

- A stream is a pipeline of CUDA instructions
- Commands issued in distinct non-zero streams may overlap, while commands in the same stream remain ordered
- As long as subsequent commands using this memory are issued in the same stream, there is no risk of using unexisting memory (even if cudaMallocAsync returns before the memory is allocated)
- CAUTION: trying to access memory allocated within one stream from another stream leads to undefined behavior, as the CUDA execution provides no guarantee for the ordering of events in different streams

Inter-Device copies

- To allow the current device (the one set by the last call to cudaSetDevice()) to directly access data on another device, one should call

```
cudaDeviceEnablePeerAccess (device_id,0);
```

- The second argument should be 0 (the documentation says that it is reserved for future evolutions of this function)
- CAUTION: This call grants a **unidirectional** permission (for the current device to access the memory of device_id). To have bidirectional access, another call to cudaDeviceEnablePeerAccess must be done with the roles of the two devices swapped

Inter-Device copies

- To perform the actual device-to-device copy, one should use

```
cudaMemcpyPeer(destination,d_device,source,s_device,size);
cudaMemcpyPeerAsync(destination,d_device,source,s_device,size,stream);
```

- The semantics of these functions is the same as that of the ordinary ones, but they need extra arguments telling the source and destination devices. They do not need a type_of_transfer because it is implicit in this context
- The source and destination pointers must have been allocated on the respective devices
- The two devices may be the same

Unified Memory

- Some devices can share a unified memory address space with other devices and with the host. Check this capability with:

```
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop,device);
if (prop.unifiedAddressing==1){
  // Device supports unified addressing
 }
```

- When the device has this capability, one may use cudaMemcpy() with the transfer type set to cudaMemcpyDefault
- To find where the memory associated to a pointer resides, use:

```
cudaPointerAttributes attr;
cudaPointerGetAttributes(&attr,pointer);
```

  - attr.cudaMemoryType can be cudaMemoryTypeHost or cudaMemoryTypeDevice
  - attr.device tells on which device the memory resides

Unified Memory

- For host allocations to work with unified memory, they should be done with the CUDA function cudaMallocHost() instead of the host's malloc().
  Host memory allocated in this fashion is directly accessible by GPUs. One can pass a host pointer directly to a CUDA kernel
- With a unified address space, after calling cudaDeviceEnablePeerAccess() appropriately, the memory allocated on a device is directly accessible by the other devices
- Memory allocated by cudaMalloc() or cudaMallocHost() remains resident on the device/host where the pointer was allocated. It is copied multiple times if requested multiple times.

  Memory allocated by cudaMallocManaged() may become resident of another device/host, and does not need to be copied multiple times when reused

CUDA kernels

What is a CUDA kernel?

- A CUDA kernel is a special type of function that maps (in parallel) a certain action onto the entries of an array
- A kernel runs on a single GPU, but its threads may be split across various *streams multiprocessors* (SM)
- The threads of a kernel are grouped in *blocks* (whose size is chosen by the programmer), and all threads in a block are guaranteed to run on the same SM (and thus see the same L1 cache, that they can use as shared memory to communicate)
- Threads in the same block can be synchronized easily, but not threads in different blocks

CUDA KERNEL DECLARATION

- CUDA kernel declaration:

```
__global__ kernel_name(kernel arguments){
  // perform some useful action here
}
```

- Kernel arguments can be scalars, structures, pointers
- The combined size of all arguments should be less than 4096 Bytes (32764 Bytes on recent GPUs)
- Each thread runs the kernel (the number of threads is chosen when the kernel is called)
- Inside the kernel, the following variables are predefined:
  - threadIdx.{x,y,z} : thread index within a block
  - blockIdx.{x,y,z} : block index
  - blockDim.{x,y,z} : block dimensions
  - gridDim.{x,y,z} : number of blocks

CUDA KERNEL DECLARATION

- In general, a multidimensional grid of threads is used as a natural way to map into a multidimensional array, as in

```
__global__ MatrixAdd(float A[N][N],float B[N][N],float C[N][N]){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  C[i][j] = A[i][j] + B[i][j];
}
```

  (a multidimensional grid makes little sense to access 1-d data)

- Size limits:
  - Max number of threads per block: 1024
  - Max number of blocks per kernel: $2^{31}-1$
  - Max number of blocks in the y and z directions: 65535

- NOTE: blocks smaller than 32 threads are not useful, because 32 is the smallest group of threads that a GPU can schedule

Kernel launch

- The general syntax for launching a CUDA kernel is

```
kernel_name<<<blocks,threads_per_block,shared_mem,stream>>>(kernel arguments);
```

- The arguments of the kernel are between the parenthesis, like for a normal function
- The parameters within $<<< \cdots >>>$ define other details of the parallel task:
  - 1st argument: number of blocks. Type: *int* or *dim3*, as in

```
int blocks = 512;           // 512 x   1 x   1
dim3 blocks(512);           // 512 x   1 x   1
dim3 blocks(512,512);       // 512 x 512 x   1
dim3 blocks(512,512,128); // 512 x 512 x 128
```

  - 2nd argument: number of threads per block, for the x, y and z directions. Type: *int* or *dim3*
  - 3rd argument: amount of shared memory per block (in bytes) used by the kernel (optional; 0 if not present)
  - 4th argument: stream in which the kernel is launched (optional; default stream if not present). Note: if a stream is indicated, then the shared memory must also be present (set to 0 if not used)

Kernel launch

- Hierarchy of threads in a CUDA kernel:
  - The set of all threads forms a *grid*
  - The grid is divided in *blocks*, whose size and number is defined when launching the kernel. All threads in the same block run on the same SM, and see the same shared memory (not true for threads that belong to different blocks)
  - The blocks do not all run simultaneously. One should not assume a particular ordering for the time at which the various blocks run
  - A block is further divided into groups of 32 threads, called *warps*. This is mostly a hardware notion (all threads in a warp execute the same instruction), but there are a few functions to do manipulations at warp level
- Simplest setup: the block size is a multiple of 32, and the size of the array to process is a multiple of the block size (otherwise, a test must be included inside the kernel to avoid reading/writing beyond the end of the array)

Divergent code

- A CUDA code is called *divergent* when the threads in a warp do not execute the same instructions. Example:

```
__device__ float fo(float x,float y){
  return  sinf(x*x+y*y);
}

[+ f1,...,f7: seven other functions of equivalent complexity]

__global__ void non_divergent_kernel(float *a,float *b){
  int id   = blockIdx.x * blockDim.x + threadIdx.x;
  a[id] = fo(a[id],b[id]);
}

__global__ void divergent_kernel(float *a,float *b){
  int id   = blockIdx.x * blockDim.x + threadIdx.x;

  switch (threadIdx.x % 8) {
  case 0:
    a[id] = fo(a[id],b[id]);
    break;
    [...]
  }
}
```

Divergent code

- Effect of code divergence on the running time:

```
non divergent code : 4.61 seconds
divergent code     : 8.61 seconds
```

- The running time is not multiplied by 8 in this example, because not all instructions are different between the threads (and the compiler is smart enough to see the common instructions)
- Sometimes, the compiler is able to know the outcome of a test at compilation time
- The compiler also tries to rearrange the code to avoid branches
- Best practice: avoid branches in kernel code

Lecture III

Outline

- Architecture of GPUs
- What tasks are GPUs good at?
- Overview of CUDA

- Diagnosing errors
- Memory management (allocation, copy)
- CUDA kernels

- Streams, Synchronization
- Shared memory
- Reduction operations

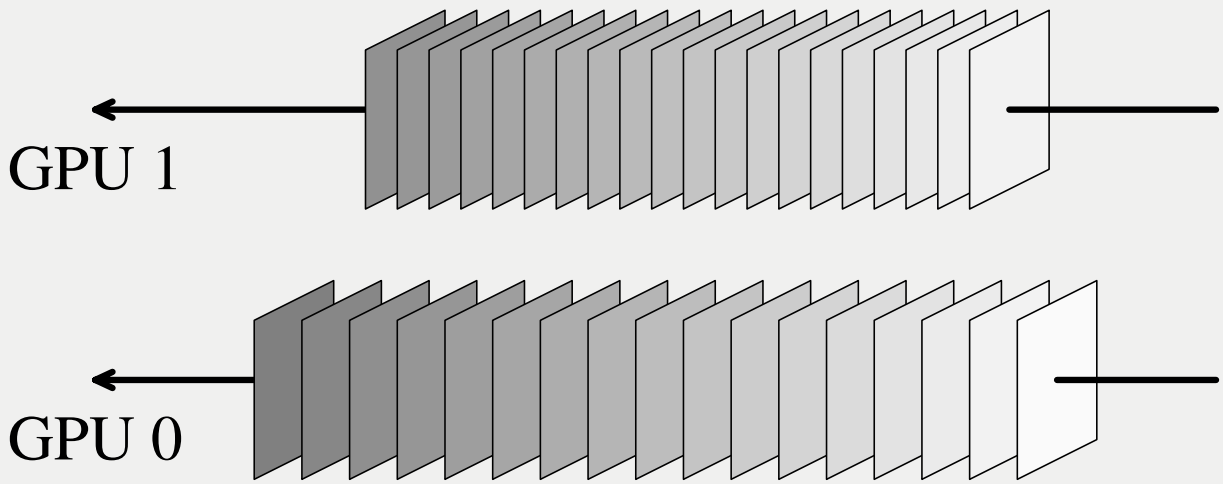
- Memory performance tuning
- How many threads?
- CUDA libraries for common tasks: cuFFT, cuBLAS, ...

Streams, Synchronization

Default stream

- All CUDA commands are queued in a pipeline, and are executed in order (First In First Out). For instance, if a kernel launch follows a memory copy, this queuing mechanism ensures that the data has been effectively copied before doing computations with it in the kernel
- Such a queue of CUDA tasks is called a stream
- By default, there is a single stream per GPU, called the default stream (or stream 0). Tasks in the default streams of distinct GPUs are not synchronized
- It is possible to have one default stream per GPU and *per host thread*. This behavior is obtained by giving the option –default-stream=per-thread to nvcc

Default stream



- CUDA only orders tasks that belong to the same stream
- If a task on GPU 1 depends on the completion of a task on GPU 0, a stronger synchronization is needed

CUDA EVENTS

- CUDA *events* are a tool to enable a host thread to wait for certain tasks to complete
- Typical usage:

```
cudaEvent_t event;
cudaEventCreate(&event);

[A: sequence of CUDA commands]
cudaEventRecord(event,stream);
[...]
cudaEventSynchronize(event);
[B: CUDA commands that should wait until A has completed]
```

- cudaEventRecord(event,stream) records in event all the pending tasks in stream on the current GPU
- cudaEventSynchronize(event) waits until all the tasks recorded in event have finished
- NOTE: if A and B happen in the same stream, this is not necessary (the stream ordering will do exactly this)

CUDA EVENTS

- CUDA events can also be used to time some tasks:

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

cudaEventRecord(start,0);
[do some stuff]
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&duration,start,stop);
```

- After this, the variable duration contains the time elapsed between start and stop, in milliseconds

- NOTE: for memory bound tasks, a better measure of the code efficiency is the effective memory bandwidth:

$$10^{-6} * (\text{reads} + \text{writes [Bytes]})/(\text{duration [msec]}) \quad \text{GB/sec}$$

User created streams

- For situations where one has several independent sets of tasks (i.e., tasks from different sets are independent, but tasks in the same set are dependent), it is possible to create additional streams
- Syntax to create a new stream on the current GPU:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
```

- The variable stream can then be used in all the ...Async commands that take a stream as argument, to indicate that the command should be issued in that stream

User created streams

- When using only the default stream, multiple kernels cannot overlap, even if the resources necessary for them would be available. Also, kernels and memory operations cannot overlap
- When issued in different streams, two kernels can be executed simultaneously (if the GPU has enough resources for both)
- Even in situations where the resources are insufficient for running two concurrent kernels, memory operations can overlap with kernel executions, if they are issued in different streams

Rules of stream synchronization

- The *legacy* default stream synchronizes with user created streams on the same GPU. For instance, if one issues

```
kernel_1<<<1,N,0,s>>>(...); // kernel 1 launched in stream s
kernel_2<<<1,N>>>(...);     // kernel 2 launched in default stream
kernel_3<<<1,N,0,s>>>(...); // kernel 3 launched in stream s
```

  kernel_2 will block until kernel_1 has finished, and kernel_3 will block until kernel_2 has finished

- The *per-thread* default streams do not synchronize with user created streams (they behave like one of them)

- User created streams do not synchronize with one another

- cudaStreamSynchronize(stream) causes the calling host thread to block until all activity in stream has finished

Synchronization among threads

- There are situations where one would like to wait until certain threads in the same kernel have reached a certain point in the calculation, before continuing
- NOTE: all threads in a warp execute the same instruction. They should be always synchronized
- __syncthreads() halts the execution until all threads *in a block* have reached this point
- It can be used for instance to make sure that all threads have written to shared memory, before reading from it
- CAUTION: __syncthreads() should not be called within a conditional branch unless one is certain the condition will be true for all threads in the block. Otherwise, certain threads will never reach the __syncthreads() and the program will hang indefinitely

Synchronization among threads

- **Block-level** synchronization functions:
  - void __syncthreads():
    halts until all threads in the block have reached it
  - int __syncthreads_count(int condition):
    halts until all threads in the block have reached it, and returns the number of threads for which condition is true
  - int __syncthreads_and(int condition):
    same, and returns non-zero if condition is true in all threads
  - int __syncthreads_or(int condition):
    same, and returns non-zero if condition is true in at least one thread

Synchronization among threads

- **Warp-level** synchronization:
  - void __syncwarp(unsigned mask): halts until all threads designated by mask have reached it. mask is a 32-bit unsigned integer, and participating threads are indicated by setting the corresponding bit to 1 (for all threads, mask=0xffffffff)
- **Warp-level** "vote" functions (do not imply a memory barrier):
  - int __all_sync(unsigned mask, condition): returns true if condition is true in all threads designated by mask
  - int __any_sync(unsigned mask, condition): returns true if condition is true in at least one thread designated by mask
  - int __ballot_sync(unsigned mask, condition): returns an integer whose bits equal to 1 indicate the threads designated by mask for which condition is true

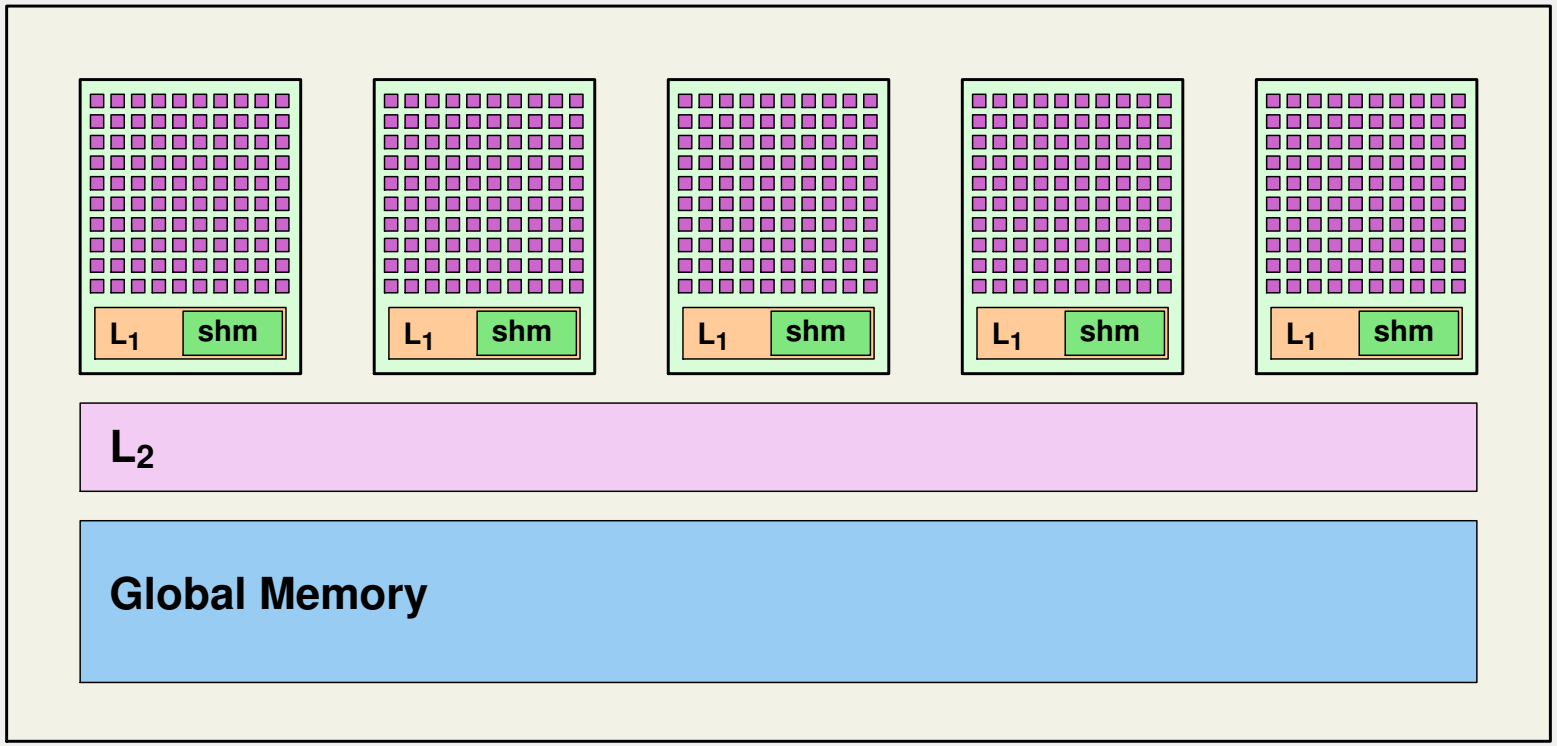
Synchronization among threads

- The synchronization of pools of threads larger than a block is problematic, since CUDA does not even promise they run simultaneously on the GPU

- When kernel-wide synchronization is necessary, the easiest method is to split the kernel in two, at the place where all threads should synchronize (the termination of a kernel is a point where all threads effectively converge)

Shared memory

What is shared memory?



- Shared Mem = part of the L1 cache that can be managed by the program
- Since all threads in a block run on the same SM, they see the same shared memory. Shared memory can be used for communication between these threads

How to declare shared memory?

- Shared memory is declared for each kernel independently
- Its amount is specified *per block* (but is the same for all blocks)
- Two ways to declare it:
  - Static declaration inside the kernel definition:

```
__global__ kernel_name(arguments){
  __shared__ float A[N];   // statically declared size
  [...]
}
```

  - Dynamic declaration in the kernel call:

```
__global__ kernel_name(arguments){
  extern __shared__ float A[];   // unspecified size
  [...]
}

kernel_name<<<blocks,blocksize,N*sizeof(float),stream>>>(arguments);
```

- Lifetime = until the block finishes

How to declare shared memory?

- If the size of shared memory is known at compile time, use a static declaration
- When the amount of shared memory is not known at compile time, or should change in various kernel launches, use a dynamic declaration
- After it has been declared one way or the other, the use of shared memory is identical in both cases
- CAUTION: asking too much shared memory limits the number of blocks that can run simultaneously on a given SM

Shared memory properties

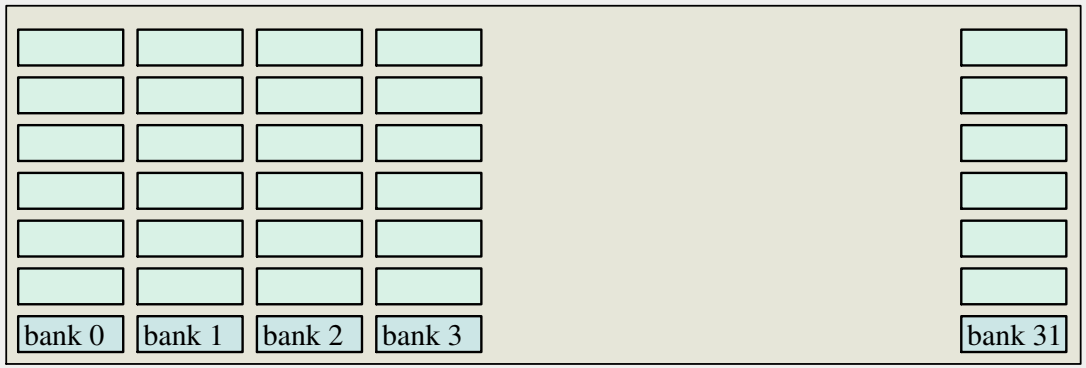
- On the same chip as the computing cores: very fast access. Can be used to fasten certain algorithms
- Size limit: 48 kBytes/SM
- Coherent for all threads that belong to the same block
- Writes to shared memory done by different threads are not guaranteed to be completed all at the same time. A *memory fence* is mandatory before one can safely read (when the writes and the reads are done by different threads):

```
[writes]
__syncthreads();
[reads]
```

__syncthreads() forces all threads in the block to halt until all have reached this function (this call should not be placed inside a conditional branch that may not be reached by all threads)

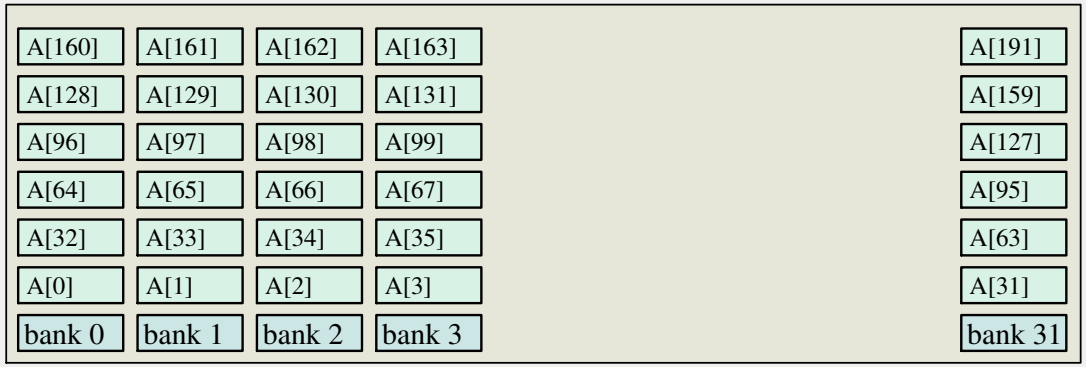
Shared memory bank conflicts

- For maximizing throughput, shared memory is organized in 32 "banks" (hardware units), each able to serve one 4-byte request during the same memory operation



Shared memory bank conflicts

- In shared memory, the successive entries of an array $A[i]$ of floats are stored in successive memory banks



- 32 consecutive entries requested by the 32 threads in a warp can thus be retrieved in a single operation

- This also works for non consecutive entries, as long as they are pulled from 32 distinct banks

Shared memory bank conflicts

- CAUTION: concurrent requests to the same memory bank (called a *bank conflict*) are serialized (except when several threads request exactly the same address)
- 32 successive threads accessing $A[0], A[1], A[2], A[3], \cdots$ have their requests fulfilled in the same memory read
- 32 successive threads accessing $A[0], A[32], A[64], A[96], \cdots$ need 32 distinct memory reads to have their requests fulfilled
- In order to maximize throughput, one should think about the access pattern to shared memory, and modify the code to minimize bank conflicts (example: 32×32 matrix, arranged in row-major order. Accessing a row of this matrix is fast, but accessing a column is slow)

Reduction operations

What is a reduction?

- Consider a large array $A[i]$ with N elements. A typical example of reduction is to compute the sum of all its elements

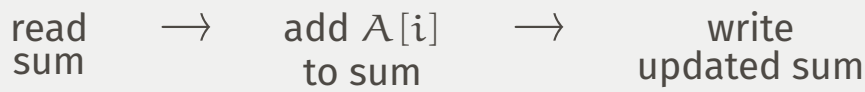
$$S \equiv \sum_{i=0}^{N-1} A[i]$$

  (additions may be replaced by some other arithmetic operation, or by a logical operator – the operation must be associative)

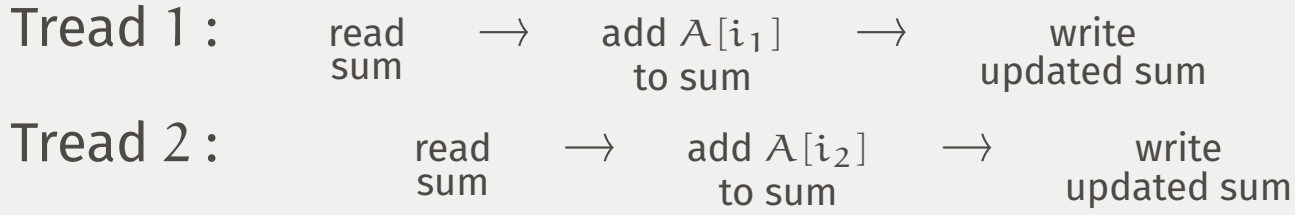
- The naive implementation of this is not parallel, since all the additions are done sequentially (there are $N-1$ additions)

Main danger with multithreaded reductions

- Each thread does a sequence of 3 actions:



- With two threads, one could have:



  i.e., Thread 2 reads the value of sum before Thread 1 has finished its update (and therefore Thread 2 gets the old value, and its update will overwrite that of Thread 1)

- To avoid this, the [read-add-write] sequence should be treated as a single insecable instruction

Naive CUDA reduction (don't do this!)

- The naive implementation uses atomicAdd() to make sure threads do not step on each other while updating the sum:

```
__global__ void grid_sum_naive(const float *A,float *sum){
  int id    = blockIdx.x * blockDim.x + threadIdx.x;
  atomicAdd(sum,A[id]);
}
```

- Timings (for 512×10^6 elements, repeated 256 times, on my laptop – host version uses 6 OpenMP threads):

```
reduction on HOST       :  32.03 seconds, sum= 7616.225586
naive reduction on GPU  : 283.42 seconds, sum= 7618.101074
```

- This naive implementation just serializes the 512×10^6 additions $\Longrightarrow$ very slow

Optimal parallel reduction

- Assume $N = 2^p$
- Divide the array into 2^{p-1} pairs of elements
- Sum the two elements in each pair, in parallel $\Longrightarrow$ 1 step to do 2^{p-1} partial sums
- Sum pairwise these partials sums $\Longrightarrow$ 1 step to do 2^{p-2} partial sums
- ...
- In total, there are p steps $\Longrightarrow p = \log_2(N)$ steps to sum N elements (this is the lower limit; no algorithm can beat this)
- NOTE: since CUDA threads in different blocks cannot communicate, this level of parallelism is not achievable on GPUs

A (partly) parallel reduction in CUDA

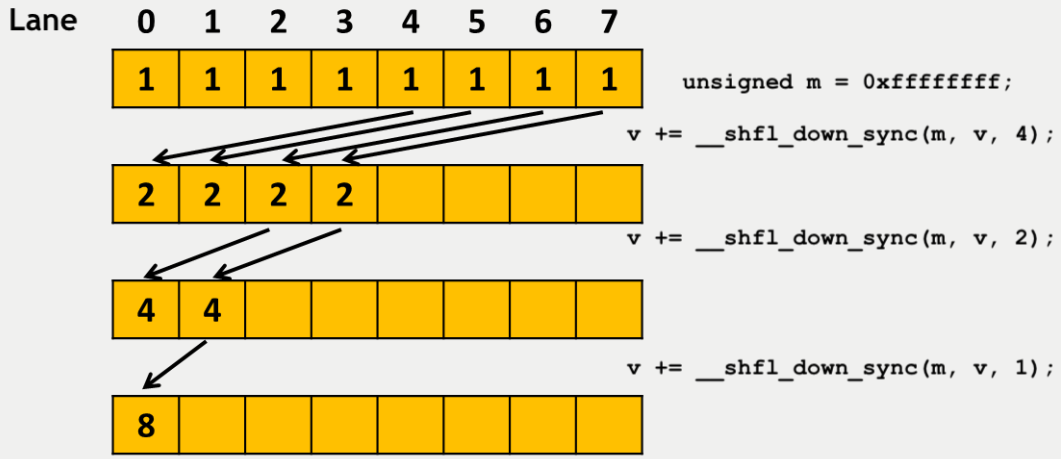
- Assign one thread per entry of the array
- Do in parallel partial reductions over each warp (warp = group of 32 threads)
- Do in parallel partial reductions over each block (using shared memory)
- Sum over all blocks (sequential)

Warp-level reduction

- Function called by the 32 threads in the warp, with the value it owns. The function returns the warp-level sum in thread 0:

```
__inline__ __device__ float warpReduceSum(float val) {
  for (int offset = 16; offset > 0; offset >>= 1)
    val += __shfl_down_sync(0xffffffff, val, offset);
  return val;
}
```

- Illustration of the last 3 steps:



(for 32 threads, there are 5 steps: offset$= 16, 8, 4, 2, 1$)

Block-level reduction

- Function called by all threads in block:

```
__inline__ __device__ float blockReduceSum(float val, int tid) {
  static __shared__ float temp[32]; // Shared mem for 32 partial sums (32 = max warps)
  int lane = tid % 32;              // thread id within the warp
  int  wid = tid / 32;              // warp id

  val = warpReduceSum(val);         // Each warp performs partial reduction
  if (lane==0) temp[wid]=val;       // thread 0 of each warp writes warp-level sum
  __syncthreads();                  // Wait for all partial reductions to complete

  if (wid==0) {                     // Done only by warp 0
    val = (tid < BLOCKSIZE / 32) ? temp[lane] : 0.0f;
    val = warpReduceSum(val);       // Call again warp-reduction to sum over the blocks
  }

  return val;
}
```

- Computes the block-level sum in 10 iterations (note: $2^{10} = 1024$)
- __syncthreads() between writes and reads to shared memory
- Returns the block-level sum in thread 0

Full-grid reduction

- In order to sum the elements of an array (whose size is a multiple of 1024), one should use the following kernel:

```
__global__ void grid_sum(const float *A,float *sum){
  int id    = blockIdx.x * blockDim.x + threadIdx.x;
  float tmp = A[id];
  __syncwarp();
  tmp = blockReduceSum(tmp,threadIdx.x);
  if (threadIdx.x==0) atomicAdd(sum,tmp);
}
[....]
cudaMemset(&sum,0,sizeof(float));
grid_sum<<<SIZE/1024,1024>>>(A,&sum);
```

- atomicAdd(a,b) adds b to a in a thread-safe way (the *read-add-store* sequence cannot interfere with other threads trying to do the same thing)
- When this kernel terminates, the variable sum (on the GPU) contains the sum of all elements of A
- Number of steps: $(\text{SIZE}/1024) * 10$ instead of $\text{SIZE} - 1$

Example: sum of 512.10^6 floats

- Sum of the elements of an array of 2^{29} floats (2 GBytes), repeated 256 times
- Parallelized with 6 OpenMP threads on host
- Parallelized with 2^{29} threads in CUDA, with blocksize=1024
- Timings (on my laptop):

```
reduction on HOST       :  32.03 seconds, sum= 7616.225586
naive reduction on GPU  : 283.42 seconds, sum= 7618.101074
reduction on GPU        :   8.04 seconds, sum= 7616.404297
```

  - Speedup by a factor ~ 4 compared to HOST
  - Relative difference $\sim 1.7 \times 10^{-5}$ (after 512 million additions)

Reduction for the logical AND

- This method is applicable to other types of reduction. For instance, finding if a certain condition is true in all threads corresponds to the logical AND of the conditions in each thread

- Warp-level reduction:

```
__inline__ __device__ unsigned warpReduceAND(unsigned val) {
  for (int offset = 16; offset > 0; offset >>= 1)
    val = val && __shfl_down_sync(0xffffffff, val, offset);
  return val;
}
```

Reduction for the logical AND

- Block-level reduction:

```
__inline__ __device__ unsigned blockReduceAND(unsigned val, int tid) {
  static __shared__ unsigned tmp[32];
  int lane = tid % 32;              // thread id within the warp
  int  wid = tid / 32;              // warp id

  val = warpReduceAND(val);         // Each warp performs partial reduction
  if (lane==0) tmp[wid]=val;        // thread 0 of each warp writes warp-level sum
  __syncthreads();                  // Wait for all partial reductions to complete

  if (wid==0) {                     // Done only by warp 0
    val = (tid < BLOCKSIZE / 32) ? tmp[lane] : 1;
    val = warpReduceAND(val);       // Call again warp-reduction to sum over the blocks
  }

  return val;
}
```

Reduction for the logical AND

- Full grid reduction:

```
__global__ void grid_AND(const unsigned *A,unsigned *and){
  int id    = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned tmp = A[id];
  __syncwarp();
  tmp = blockReduceAND(tmp,threadIdx.x);
  if (threadIdx.x==0) atomicAND(sum,tmp);
}
[....]
unsigned h_and=1; // 1 is the neutral value for AND
cudaMemcpy(&and,&h_and,sizeof(unsigned));
grid_sum<<<SIZE/1024,1024>>>(A,&and);
```

Lecture IV

Outline

- Architecture of GPUs
- What tasks are GPUs good at?
- Overview of CUDA

- Diagnosing errors
- Memory management (allocation, copy)
- CUDA kernels

- Streams, Synchronization
- Shared memory
- Reduction operations

- Memory performance tuning
- How many threads?
- CUDA libraries for common tasks: cuFFT, cuBLAS, ...

Memory performance tuning

Pointer aliasing

- In C/C++, pointers to variables of compatible types may *alias*, i.e., point to the same (or overlapping) areas in memory
- Consider a function, or CUDA kernel, declared as

  ```
  void function_name(float *a, float *b, float *c)
  ```

  that contains the following instructions

  ```
  c[0] = a[0] + b[0];
  c[1] = a[0] + b[1];
  [...]
  c[9] = a[0] + b[0];
  ```

- If the pointer c refers to memory that overlaps with a or b, writes to $c[\cdot]$ can in fact modify the content $a[\cdot]$ or $b[\cdot]$
- When aliasing is possible, the compiler cannot reuse data previously loaded from memory, nor the result of seemingly identical sub-expressions. It must reload data from memory and recompute these expressions every time

Pointer aliasing

- The compiler assumes that the programmer obeys the aliasing rules strictly (if the programmer does not, the program may misbehave)
- It is illegal to access some data via a pointer to an incompatible type. Doing so leads to undefined results, as in

```c
#include <stdio.h>

int func(float *f,int *i){
  *i=1;   // the compiler can change the order of the assignments
  *f=0.0; // because the two pointers refer to different types
  return *i;
}

void main(){
  int x = 0;
  x = func((float *)(&x),&x); // the pointers refer to the same memory!!
  printf("x = %d\n",x);  // prints 0 without optimization and 1 with -O2
}
```

- Exception: char* can alias with any other pointer

Pointer aliasing: using __restrict__

- One may tell the compiler that the pointers DO NOT alias by modifying the function declaration as follows

```
void function_name(float * __restrict__ a,
                   float * __restrict__ b,
                   float * __restrict__ c)
```

- __restrict__ is a promise that the programmer makes to the compiler: *the data corresponding to a pointer decorated with __restrict__ is not read/written via another pointer*

  CAUTION: Failure to honor this promise may lead to a malfunctioning program

- When the pointers are qualified with __restrict__, the compiler is free to reuse sub-expressions and to store previously loaded data in registers

Coalesced memory access

- Thanks to the way the cache works, 32 reads/writes to 32 successive addresses in memory are done in a single operation (these accesses are said to have *coalesced*)
- This pattern of access to global memory is the one that gives the highest throughput
- The GPU will satisfy the reads/writes performed by the threads in a warp in as few memory transactions as possible
- The worst situation is that of a totally random pattern of memory access, or reading/writing data with large jumps

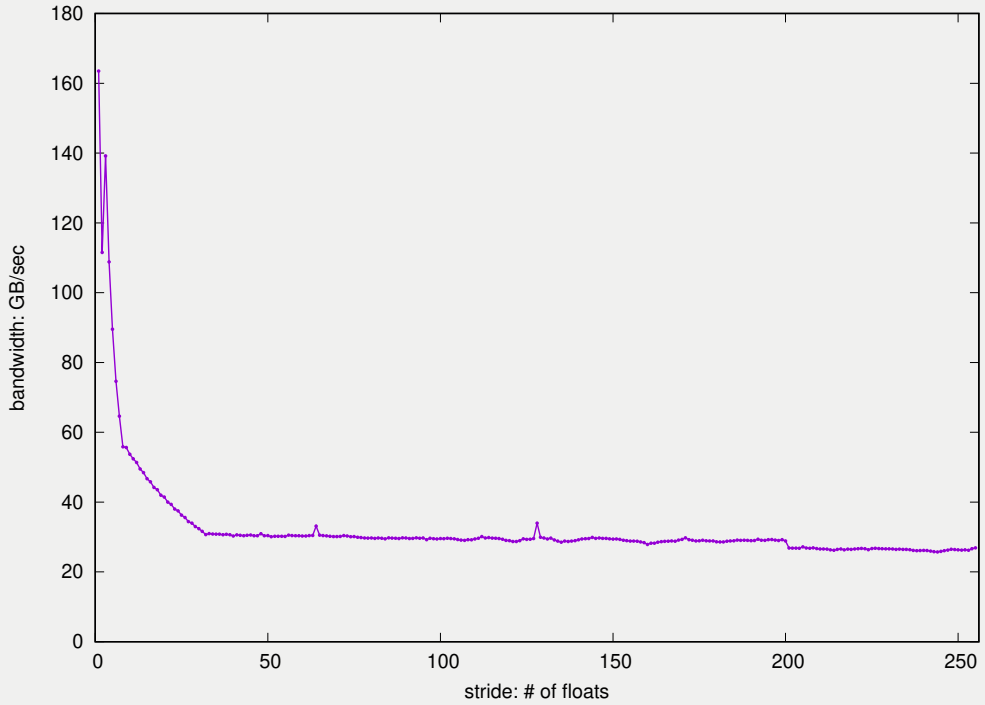
Coalesced memory access: effect of a stride on the bandwidth

- CUDA kernel that copies data from an array into another array, with a gap between the elements that are copied:

```
__global__ void copy(float *out, const float *in,int stride){
  int i = blockIdx.x * blockDim.x + stride * threadIdx.x;
  out[i] = in[i];
}
[...]
int stride = 1;
copy<<<nx/256-stride,256>>>(d_out,d_in,stride);
```

- Copies one float every stride from in[] to out[]
- Measure the time to copy and infer the bandwidth
- Vary stride from 1 to 256, and plot bandwidth versus stride

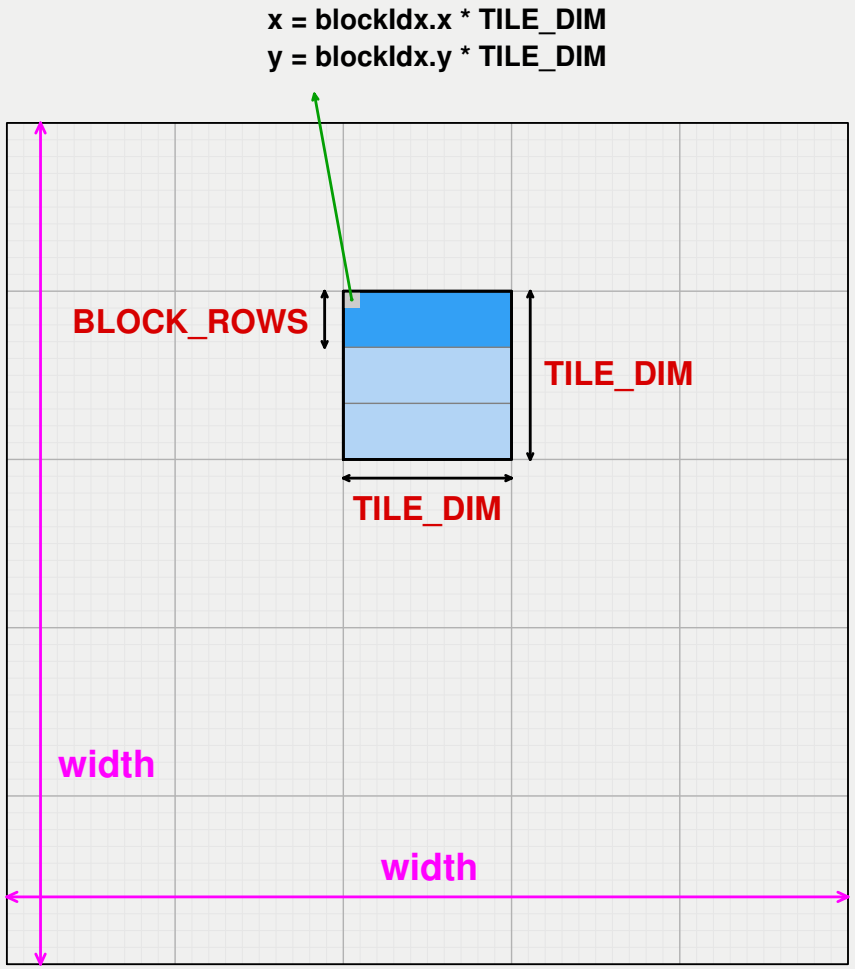
Coalesced memory access: effect of a stride on the bandwidth



- A stride between successive reads/writes implies that all threads in a warp cannot be satisfied with a single cache line
- Coalesced memory operations should be the goal for any memory-bound computation

Coalesced memory access – Example: matrix transpose

- We want to compute the transpose of a large matrix
- The width $\times$ width matrix is decomposed in square tiles:



Coalesced memory access – Example: matrix transpose

- Each block of threads handles one tile
- Naive CUDA implementation:

```
__global__ void transposeNaive(float *odata, const float *idata){
  int x     = blockIdx.x * TILE_DIM + threadIdx.x;
  int y     = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x  * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[x*width + (y+j)] = idata[(y+j)*width + x];
}
```

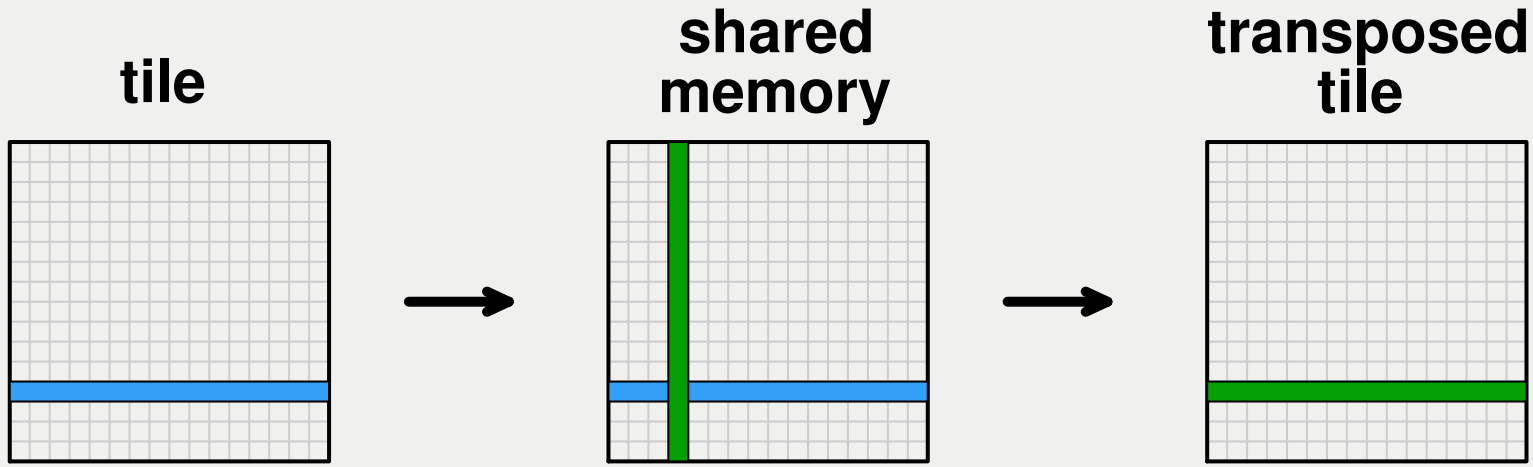
- Matrix transpose is a memory bound operation. To measure performance, we compare the effective bandwidth to that of a simple matrix copy:

```
                plain  copy :          159.57 GB/sec
        shared memory copy :          168.69 GB/sec
           naive transpose :           35.17 GB/sec
```

  Main problem: in the transpose, one of the matrices is accessed with rows and columns swapped $\Longrightarrow$ big gaps between the accessed elements $\Longrightarrow$ uncoalesced accesses

Coalesced memory access – Example: matrix transpose

- Use an intermediate buffer *in shared memory*, so that all reads/writes to global memory have coalesced:



NOTE: this works if reading along the columns in shared memory is as fast as writing along the rows

Coalesced memory access – Example: matrix transpose

- First copy the elements of a tile to an array in shared memory, following rows to benefit from coalesced reads
- Use this array in shared memory to perform the transpose

```
__global__ void transposeCoalesced(float *odata, const float *idata){
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x     = blockIdx.x * TILE_DIM + threadIdx.x;
  int y     = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x]; // coalesced reads to ida

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
    odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j]; //coalesced writes to c
}
```

Shared memory bank conflicts – Example: matrix transpose

- Significant improvement, but not yet on par with plain copy:

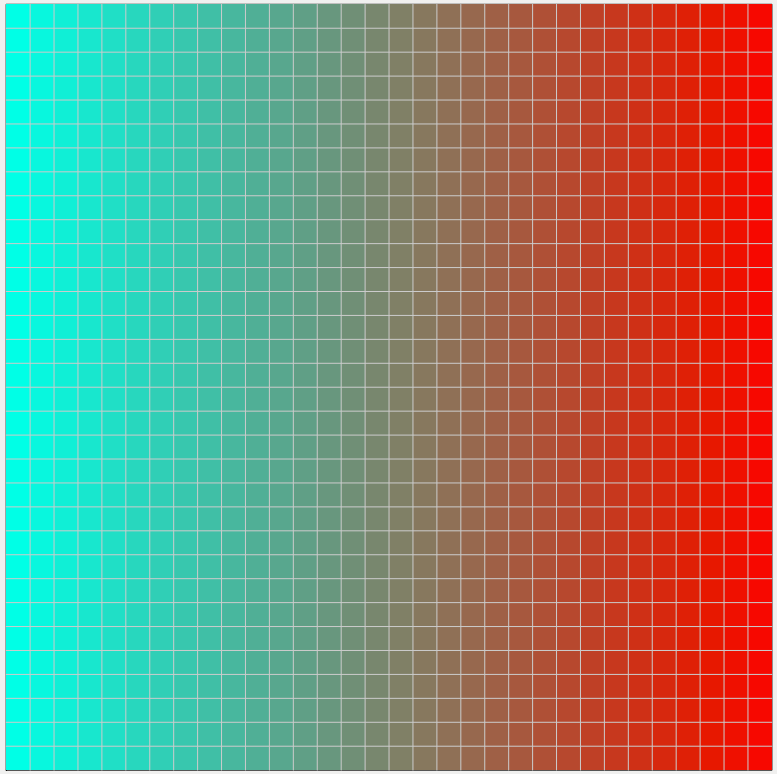
```
                   copy :            159.57 GB/sec
         shared memory copy :            168.69 GB/sec
            naive transpose :             35.17 GB/sec
        coalesced transpose :             65.53 GB/sec
```

- Penalty when concurrent reads happen in the same bank of shared memory

- In this implementation, TILE_DIM=32. Therefore, all elements in the same column of the tile fall in the same bank!

Shared memory bank conflicts – Example: matrix transpose

- **tile[32][32]** (color shades indicate memory banks):



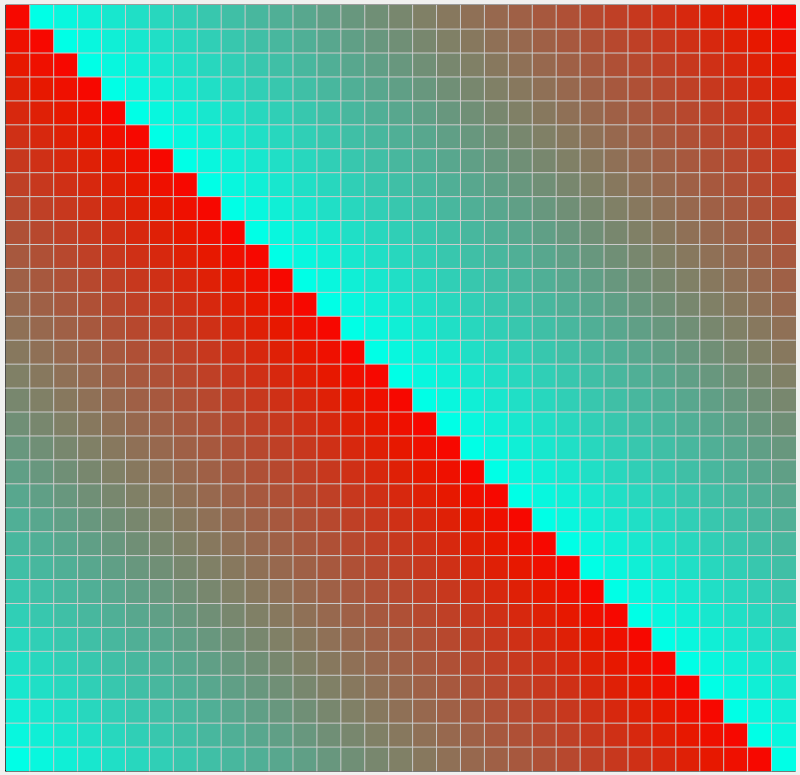
Shared memory bank conflicts – Example: matrix transpose

- Easy fix: increase one dimension of the shared array from TILE_DIM to TILE_DIM+1:

```
__global__ void transposeNoBankConflicts(float *odata, const float *idata){
  __shared__ float tile[TILE_DIM][TILE_DIM+1];   // NOTE the +1 (only change)

  int x     = blockIdx.x * TILE_DIM + threadIdx.x;
  int y     = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Shared memory bank conflicts – Example: matrix transpose

- **tile[32][33]** (color shades indicate memory banks):



NOTE: the 33rd (last) column is not used

Shared memory bank conflicts – Example: matrix transpose

- Now transpose bandwidth comparable with plain copy:

```
                    copy :            159.57 GB/sec
          shared memory copy :        168.69 GB/sec
             naive transpose :         35.17 GB/sec
         coalesced transpose :         65.53 GB/sec
    conflict-free transpose :        164.70 GB/sec
```

- Summary:
  - Aim at sequential reads/writes in global memory, with no strides
  - Use shared memory as intermediate buffer to rearrange accesses to global memory
  - Pay attention to bank conflicts when using shared memory

How many threads?

Global memory latency

- Latency between a request of data in global memory and the moment this data becomes available to the computing cores $\sim$ 500 clock cycles

- How does a GPU cope with this?

  The GPU maintains several lists of threads:

  - Threads that have the operands ready (i.e., loaded in registers) for their next instruction
  - Threads that have their operands "in flight" (i.e., the operands have been requested to memory, but have not yet arrived)
  - Threads whose next operands have not yet been requested

- The GPU prefetches the operands long before they will be used, in order to hide the memory latency

Global memory latency

- GPUs have zero overhead for switching the active threads, and schedule threads so that the computing cores are always busy
- Long latency instructions (e.g., memory loads) are scheduled as early as possible
- For this mechanism to work, there should be enough threads so that, at any given time, the number of threads ready to run is comparable to the number of computing cores
- Recommendations:
  - not too large blocksize (128-256 threads) so that many blocks are able to run concurrently
  - have several hundred blocks at a minimum
  - at least ~ 10000 threads in total
  - help the compiler reshuffle instructions by using __restrict__

CUDA libraries

cuBLAS

What is BLAS?

- BLAS = Basic Linear Algebra Subprograms
- BLAS is a specification that defines a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication
- De facto standard for the underlying linear algebra manipulations in many softwares
- Many implementations of BLAS exist, in principle interchangeable

BLAS – PERFORMANCE CONSIDERATIONS

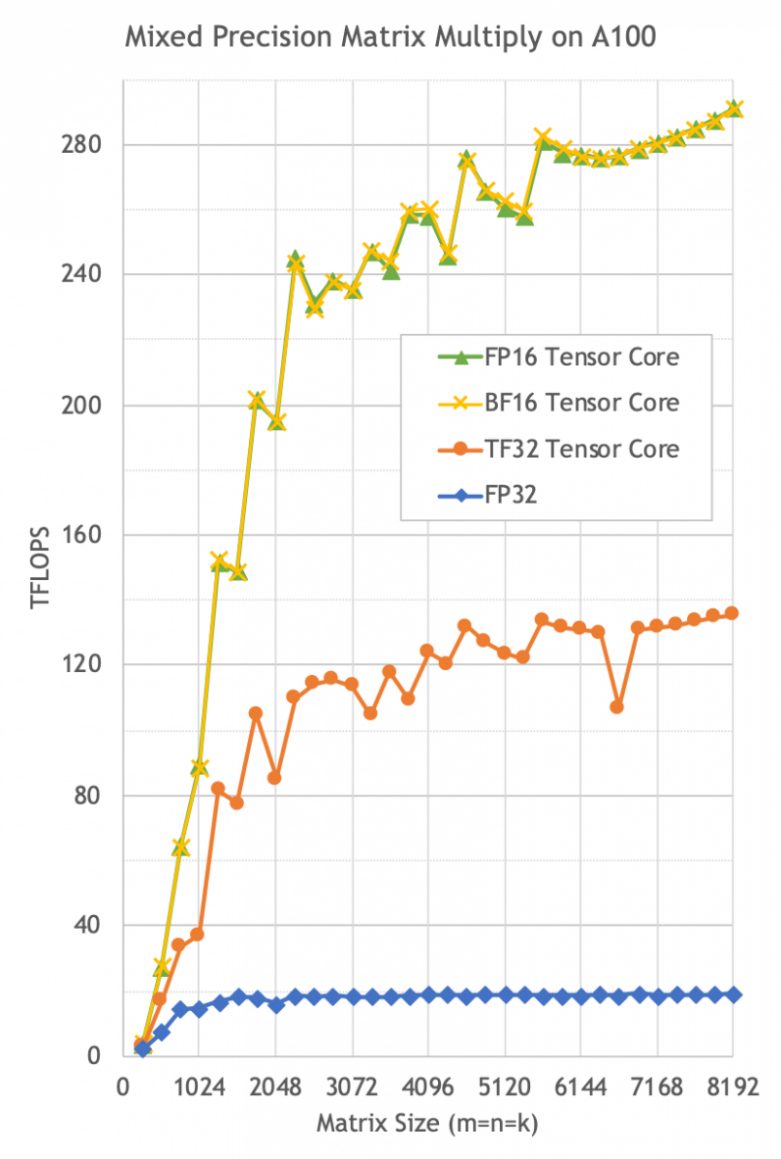
- BLAS routines often perform identical operations on the entries of large arrays $\Longrightarrow$ potential for improvement on GPUs
- Varying levels of computational intensity depending on the type of BLAS routines:
  - Vector-vector, matrix-vector:
    (computations)/(memory accesses) $\sim \mathcal{O}(1)$
  - Matrix-matrix:
    (computations)/(memory accesses) $\sim \mathcal{O}(N)$

    $\Longrightarrow$ different optimization strategies needed for both cases

cuBLAS

- cuBLAS is a GPU-accelerated library that implements the BLAS routines
- The cuBLAS library is highly optimized for performance on NVIDIA GPUs
- May use tensor cores for acceleration of low- and mixed-precision matrix multiplication

cuBLAS


Mixed Precision Matrix Multiply on A100
TFLOPS
0
40
80
120
160
200
240
280
0
1024
2048
3072
4096
5120
6144
7168
8192
Matrix Size (m=n=k)
FP16 Tensor Core
BF16 Tensor Core
TF32 Tensor Core
FP32

cuBLAS

- Header: #include <cublas_v2.h>
- The application must initialize a handle to the cuBLAS library context by calling the cublasCreate(&handle) function (handle is of type cublasHandle_t). This handle is explicitly passed to every subsequent library function call
- The library is thread safe and its functions can be called from multiple host threads, even with the same handle. BUT: some settings are stored in the handle, that should not be changed concurrently by threads
- cublasSetStream(handle,stream) tells the library to execute in stream all functions using handle (implicitly: default stream)

cuBLAS – Example of matrix multiplication

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transA,
                           cublasOperation_t transB,
                           int m, int n, int k,
                           const float *alpha,
                           const float *A, int lda,
                           const float *B, int ldb,
                           const float *beta,
                           float *C, int ldc);
```

- S for single precision (also: H,D,C,Z)
- Performs: $C = \alpha\, T_a(A) T_b(B) + \beta\, C$
- $T_a(A) = A$ if transA = CUBLAS_OP_N, $T_a(A) = A^t$ if transA = CUBLAS_OP_T , $T_a(A) = A^\dagger$ if transA = CUBLAS_OP_C
- $m =$ number of rows of A and C
- $n =$ number of columns of B and C
- $k =$ number of columns of A and rows of B
- For $\cdots$_OP_N: $A = [\text{lda} \times k]$ with $\text{lda} \geq m$; $B = [\text{ldb} \times n]$ with $\text{ldb} \geq k$
- $C = [\text{ldc} \times n]$ with $\text{ldc} \geq m$

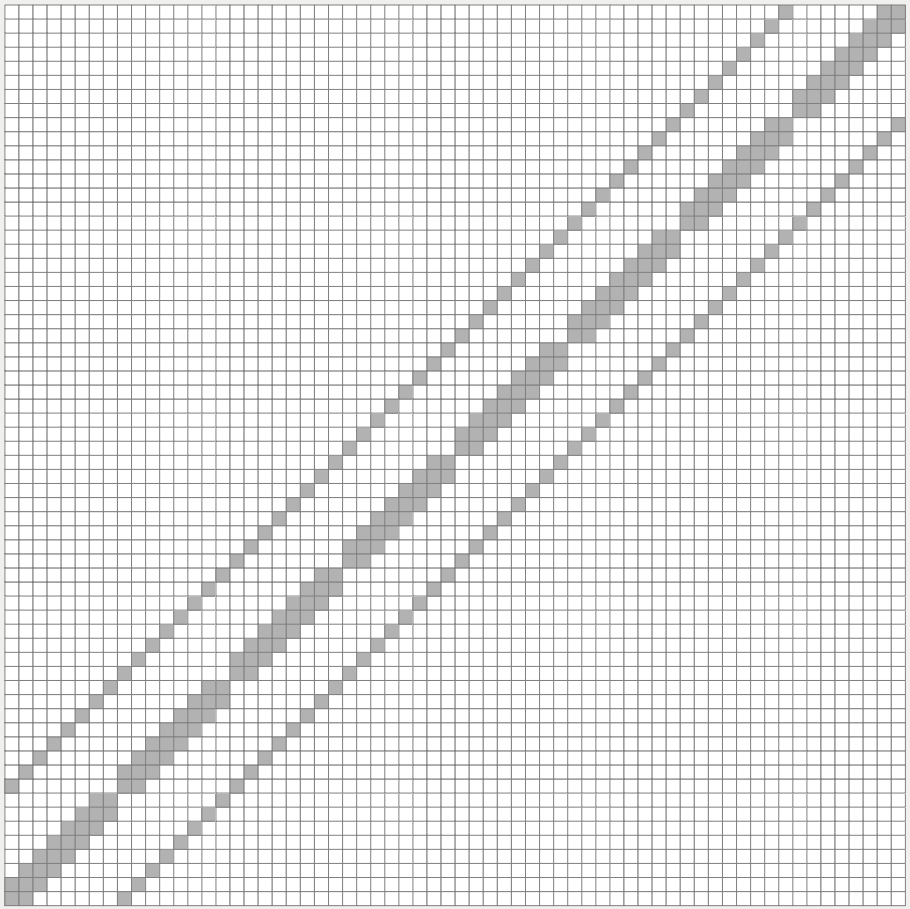
cuSPARSE

cuSPARSE

- The cuSPARSE library contains a set of GPU-accelerated basic linear algebra subroutines used for handling sparse matrices
- Sparse matrices = matrices where a large fraction of the entries are zero
- Zero entries need not be stored
- Arithmetic with zero entries is trivial
- The cuSPARSE library targets matrices with sparsity ratios in the range between $70\% - 99.9\%$

cuSPARSE

- Sparse matrices are encountered when discretizing a local operator. Example: Laplacian on a 8×8 grid



cuSPARSE

- cuSPARSE provides various types of operations:
  - Operations between a sparse vector and a dense vector
  - Operations between a dense matrix and a sparse vector
  - Operations between a sparse matrix and a dense vector
  - Operations between a sparse matrix and a dense matrix
  - Operations between a sparse matrix and a sparse matrix
  - Operations between dense matrices with output a sparse matrix
- cuSPARSE provides several storage formats (for vectors and matrices, dense or sparse)

cuSOLVER

cuSOLVER

- cuSOLVER is a GPU accelerated library for decompositions and linear system solutions for both dense and sparse matrices
- Based on the cuBLAS and cuSPARSE libraries
- Provides LAPACK-like features:
  - common matrix factorizations
  - triangular solve routines
  - least-squares solver
  - eigenvalue solver

cuFFT

cuFFT

- cuFFT is a CUDA Fast Fourier Transform library
- Algorithms highly optimized for input sizes that can be written in the form $2^a \times 3^b \times 5^c \times 7^d$. Smaller prime factor $\Rightarrow$ better performance. Powers of two are the fastest
- $\mathcal{O}(N \times \log N)$ for all input sizes N
- Half-, single- and double precision. Smaller formats are faster
- Complex and real-valued input and output
- 1D, 2D and 3D transforms
- Execution of multiple 1D, 2D and 3D transforms simultaneously
- In-place and out-of-place transforms
- FFTW compatible data layout (use cuFFTW for porting code already using FFTW)

cuFFT

- Header: #include <cufft.h>
- cufftPlanMany(&handle,· · ·): stores in handle (type: cufftHandle) the parameters of a sequence of identical Fourier transforms
- cufftSetStream(handle,stream): sets the stream in which to perform all calls associated to a given handle
- cufftExecR2C(handle,· · ·): example of a call doing an actual FFT (with real input and complex output)

cuRAND

cuRAND

- The cuRAND library provides facilities for generating pseudorandom numbers
- A pseudorandom sequence of numbers satisfies most of the statistical properties of a truly random sequence but is generated by a deterministic algorithm
- Random numbers can be generated on the device (and stored in global memory for later use) or on the host CPU
- Provides many different generators
- Can start at some offset in the sequence (to continue a previous run without overlap in the generated sequence)
- Best performance by generating blocks of random numbers that are as large as possible (as opposed to many calls that generate only one random number at a time)