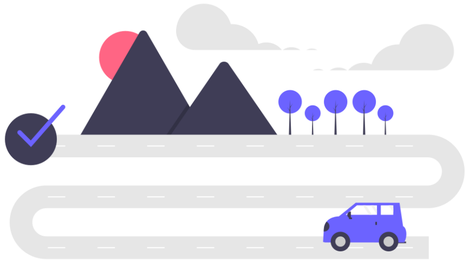
CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

"Software is like entropy: it is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics, i.e., it always increases."
(Norman Augustine)

Stanford University

Recap

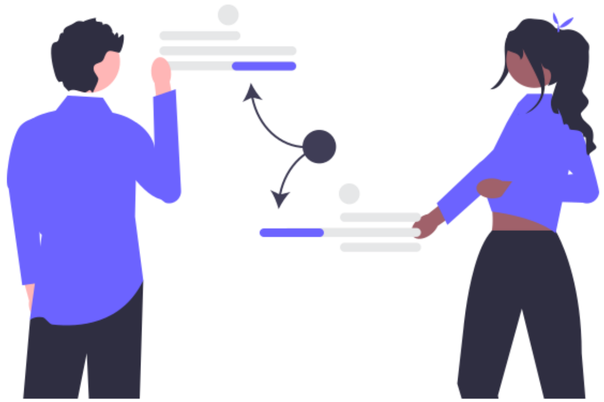


- CUDA compute capability
- Compiling and running CUDA code on icme-gpu
- Hello World CUDA code
- Warps, thread blocks, launching kernel
- Check CUDA error codes

Compiling CUDA code

The problem

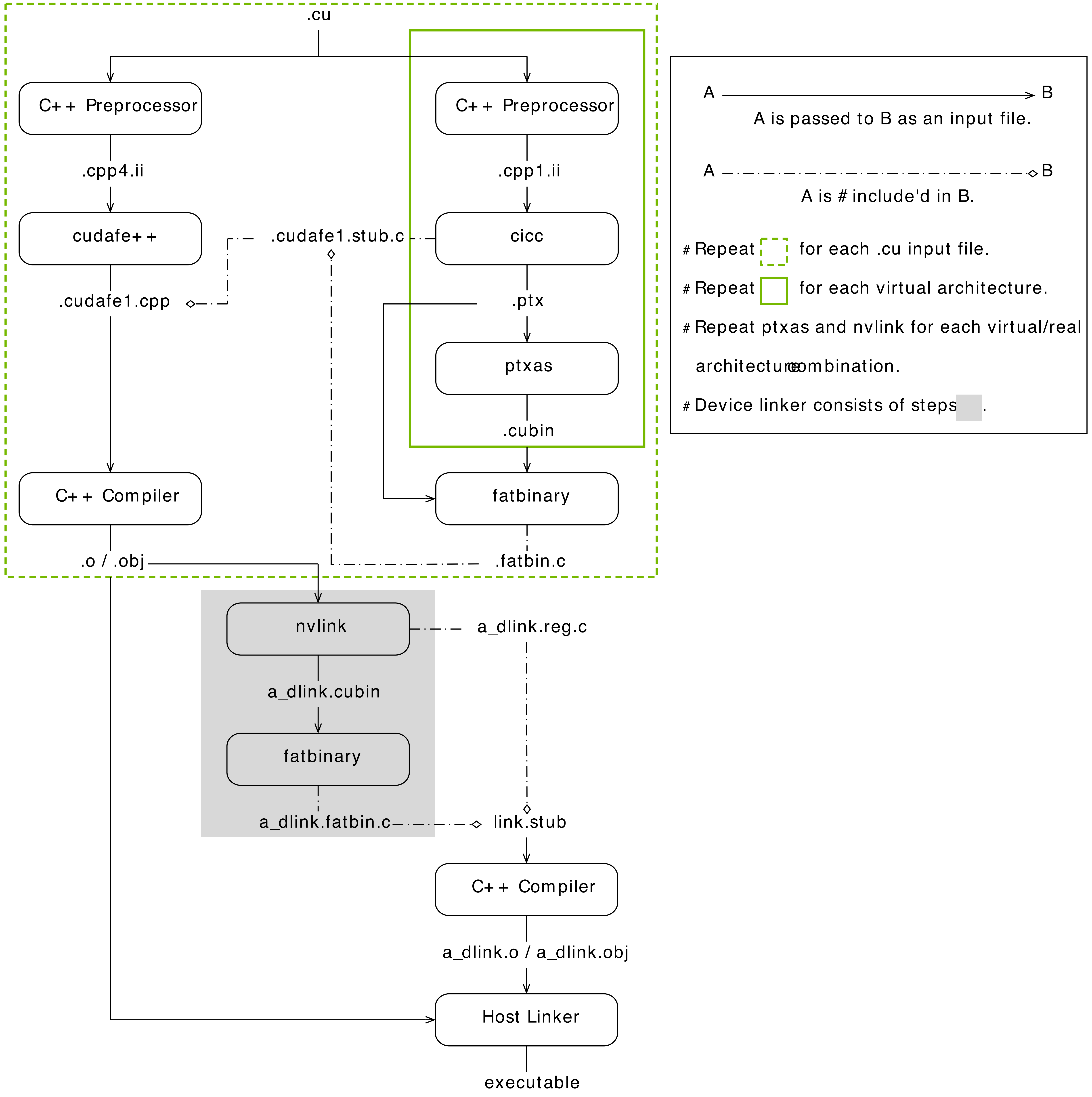
- Portability: your code should run on as many GPUs possible
- Performance: the binary should be optimized for the GPU architecture the code is running on
- There are many generations of GPUs and compute capabilities



GPU compilation trajectory

Steps in CUDA compilation:

- The input program is preprocessed and is compiled to a **CUDA binary (cubin) and/or PTX intermediate code,** which are placed in a **fatbinary.**
- The input program is preprocessed once again for **host compilation.**
- Then the C++ host compiler compiles the synthesized host code with the embedded fatbinary into a **host object.**
- The embedded fatbinary is inspected by the CUDA **runtime system** whenever the device code is launched by the host program to obtain an **appropriate fatbinary image for the current GPU.**



CUDA needs to accommodate multiple GPU architectures

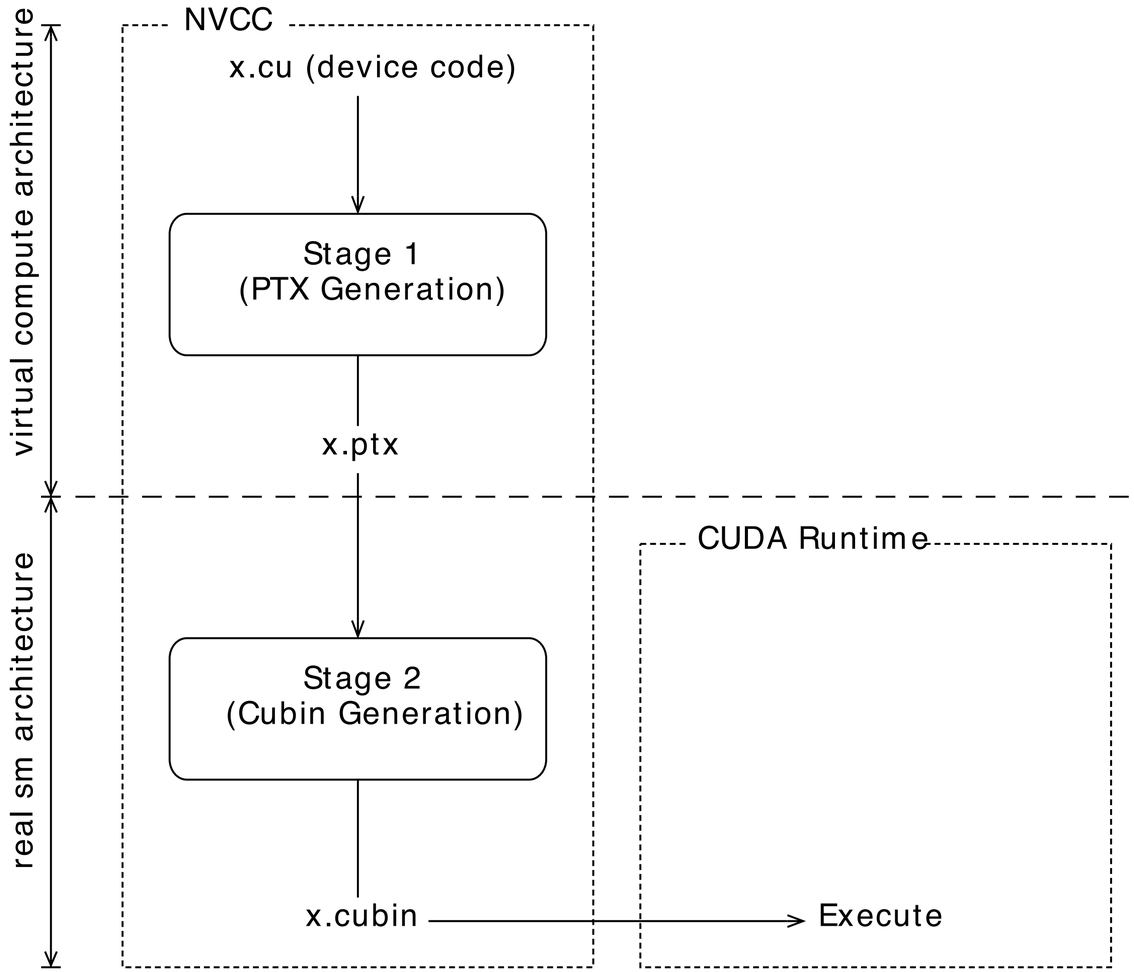
- We covered compute capabilities previously.
- We need to account for these differences when generating a GPU binary.
- Two different types of code can be generated:
  1. Code for **virtual** architecture is generated = **PTX**
  2. Code for **real architecture** is generated

PTX

- PTX (Parallel Thread Execution) assembly code relies on a specific set of features or GPU capabilities:
  - PTX = a low-level parallel thread execution virtual machine and instruction set architecture (ISA).
  - PTX exposes the GPU as a data-parallel computing device.
- Real architecture: binary code that can be executed on a given GPU

Two-Staged Compilation with Virtual and Real Architectures

- GPU compilation is performed via an intermediate representation, PTX, which can be considered as assembly for a virtual GPU architecture.
- A virtual GPU architecture provides a (largely) generic instruction set.
- An nvcc compilation command always uses two architectures: a virtual intermediate architecture, plus a real GPU architecture to specify the intended processor to execute on.
- For such an nvcc command to be valid, the real architecture must be an implementation of the virtual architecture.



Files extension zoo

File extension	Description
`.cu`	CUDA source file
`.ptx`	PTX intermediate assembly file
`.cubin`	CUDA device code binary file
`.fatbin`	CUDA fat binary file that may contain multiple PTX and CUBIN files

Choosing the right architecture

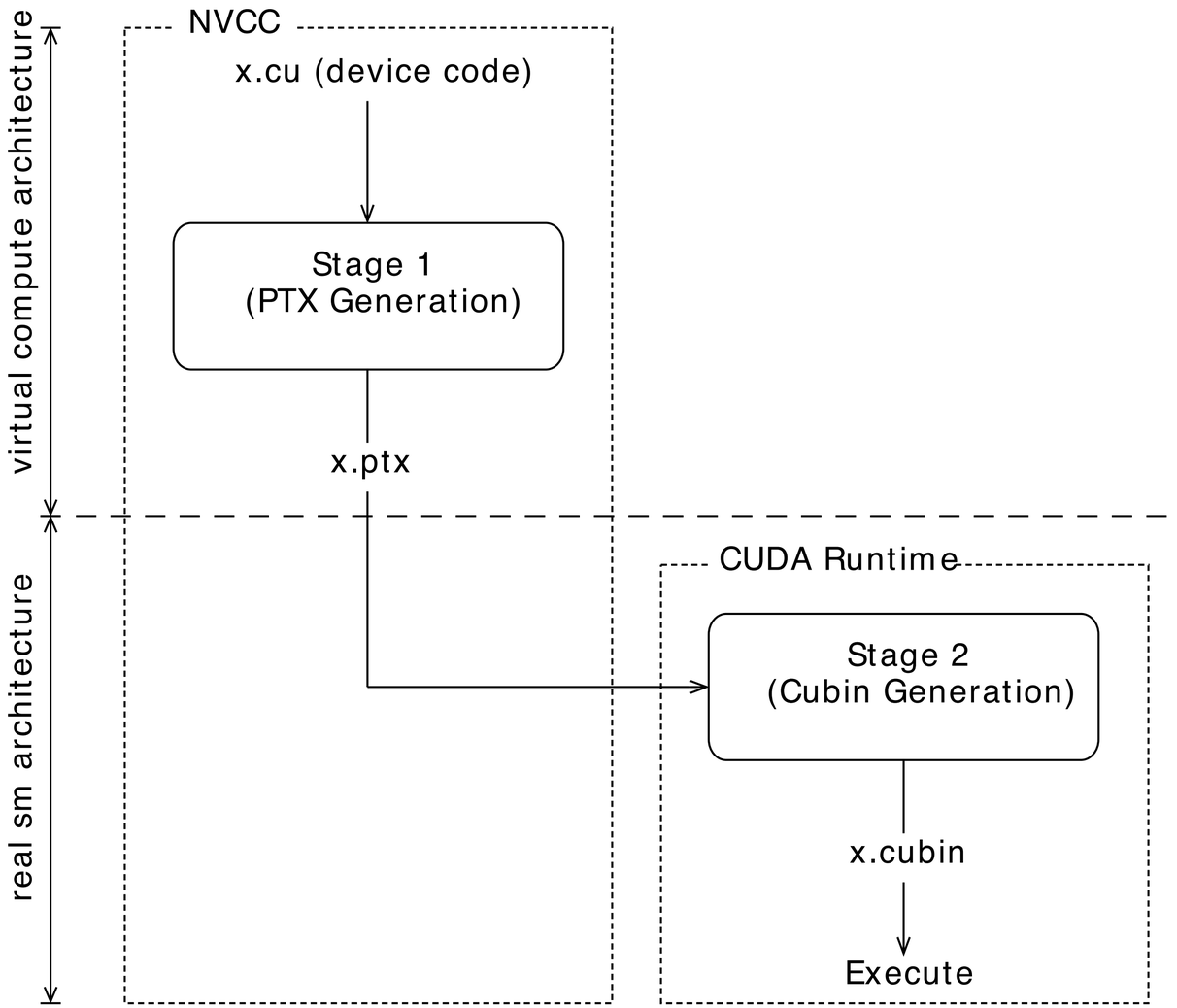
- The chosen virtual architecture is a statement on the GPU capabilities that the application requires.
- Using the **smallest** virtual architecture allows the widest range of actual architectures for the second nvcc stage.
- The real architecture should be chosen as high as possible (assuming that this always generates better code).
    - But this is only possible with knowledge of the actual GPUs on which the application is expected to run.

Table of virtual architectures

Compute capability	GPU generation
compute_35 and compute_37	Kepler support Unified memory programming Dynamic parallelism support
compute_50, compute_52, and compute_53	+ Maxwell support
compute_60, compute_61, and compute_62	+ Pascal support
compute_70 and compute_72	+ Volta support
compute_75	**+ Turing support**
compute_80, compute_86, and compute_87	+ Ampere support
compute_89	+ Ada support
compute_90, compute_90a	+ Hopper support

Just-in-Time Compilation

- If we specify a virtual code architecture instead of a real GPU, nvcc postpones the assembly of PTX code **until application runtime,** when the target GPU is exactly known.
- The disadvantage of **just-in-time compilation** is increased application startup delay.



Command line options

- When compiling with nvcc, we choose:
  - one virtual architecture,
  - some (or none) real architectures.
- If a real architecture is compiled and matches the GPU, the binary is loaded and runs!
- If a real architecture for the GPU is missing, a matching GPU binary code is generated when the application is launched using the PTX code (this is **just-in-time** compilation).

Syntax

- Virtual architecture names start with `compute_`
- Real architecture names start with `sm_`

Example

```
$ nvcc a.cu --gpu-architecture=compute_50 --gpu-code=sm_50,sm_75
```

- Use *virtual architecture* `compute_50`
- Generate code for *two GPUs:* `sm_50, sm_75`
- The real GPUs are compatible with the chosen virtual architecture.

On `icme-gpu`, try

```
--gpu-architecture=compute_50 --gpu-code=sm_50
```

Hint: icme-gpu has Turing GPUs = `compute_75`

Fails because our GPU is `sm_75`.

The generated code is for an `sm_50` GPU.

```
code=209(cudaErrorNoKernelImageForDevice)
```

Try instead

```
--gpu-architecture=compute_50 --gpu-code=sm_75
```

Success because `sm_75` matches the GPU.

Incompatible options

```
--gpu-architecture=compute_75 --gpu-code=sm_50
```

Fails because `sm_50` does not support `compute_75` features.

```
nvcc fatal: Incompatible code generation requested
```

Detailed syntax rule

```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50,sm_50,sm_52
```

`--gpu-architecture` : phase 1 compilation; virtual architecture

`--gpu-code` : phase 2 compilation; binary for each real architecture (such as `sm_50`), and PTX code for virtual architectures (such as `compute_50`; will trigger JIT if required).

Try again

```
--gpu-architecture=compute_50 --gpu-code=compute_50,sm_50
```

Works!

Why?

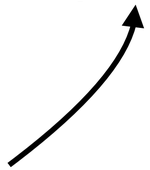
Because PTX `compute_50` can be just-in-time compiled for `sm_75` !

Shorthand 1: real architecture

```
nvcc x.cu --gpu-architecture=sm_52
```

Equivalent to:

```
nvcc x.cu --gpu-architecture=compute_52 --gpu-code=sm_52,compute_52
```



closest virtual architecture

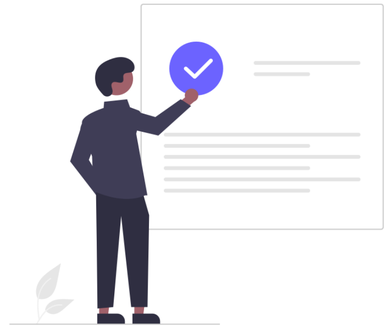
Shorthand 2: virtual architecture

```
nvcc x.cu --gpu-architecture=compute_50
```

Equivalent to:

```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=compute_50
```

References



List of virtual architectures

List of real architectures

Standard command line options for nvcc

Compiler options	Description
`-g`	Debug on the host
`-G`	Generate debug information for device code. This option turns off all optimizations on device code. It is not intended for profiling; use `--generate-line-info` instead for profiling.
`--generate-line-info (-lineinfo)`	Generate line-number information for device code.
`-pg`	Profiling info for use with gprof (Linux)
`-Xcompiler`	Options for underlying gcc compiler
`-O`	Optimization level
`--std{c++03,|c++11|c++14|c++17}`	Select a particular C++ dialect

```
nvcc --help
```

CUDA Profiler

The old way:

- Visual Profiler

NVIDIA Nsight Systems

The new way:

- NVIDIA Nsight Systems for GPU and CPU sampling and tracing
- NVIDIA Nsight Compute for GPU kernel profiling

NVIDIA Nsight Compute

CUDA-MEMCHECK

- CUDA-MEMCHECK

```
$ cuda-memcheck [options] app_name [app_options]
```

- CUDA-MEMCHECK is a **functional correctness** checking suite included in the CUDA toolkit.

- It contains multiple tools that can perform different types of checks.

  - The memcheck tool is capable of precisely detecting and attributing **out of bounds and misaligned memory access** errors in CUDA applications.

  - The tool also reports **hardware exceptions** encountered by the GPU.

CUDA-MEMCHECK tools

Tool	Description
`memcheck`	Memory access error and leak detection
`racecheck`	Shared memory data access hazard detection
`initcheck`	Unitialized device global memory access detection
`synccheck`	Thread synchronization hazard detection

GPU optimization and performance

Data movement, bandwidth, memory hierarchy

- How to optimize data transfer from GPU memory
- Caches are used to optimize memory accesses: L1 and L2 caches.
- Cache behavior is complicated and depends on the compute capability of the GPU. We will focus on Turing `sm_75`.

L1 cache

- Used for local memory (memory local to each thread) and register spills (not enough space for all the registers).
- Data that is read-only for the entire lifetime of the kernel (as determined by the compiler) can be cached in L1.
- Local to an SM.

L2 cache

- Cache accesses to local and global memory
- Shared by all SMs on the GPU
- Memory accesses that are cached in L2 only are serviced with **32-byte memory transactions**
- That's 8 float or 1 byte per thread in a warp.
- If each thread reads a float, that's 4 x 32-bytes.

Memory requests

- Each memory request from a warp is broken down into cache line requests that are issued independently.
- L1 cache line: 128 bytes
- L2 cache line: 32 bytes
- A cache line request is serviced at the throughput of the L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.

Benchmarks

We use a simple kernel that copies an array to test the L2 bandwidth:

```
__global__ void Copy(size_t n, float* odata, float* idata) {
  size_t xid = blockIdx.x * blockDim.x + threadIdx.x;
  if (xid < n) odata[xid] = idata[xid];
}
```

Memory access

- Warp requests several memory addresses.
- These are translated into cache line requests (with a granularity of 32 bytes). Then, memory requests are serviced.
- **Coalesced access:** for every 32-byte cache line, all 32 bytes are requested and used by the warp.
- This is the most efficient access pattern.

Aligned access

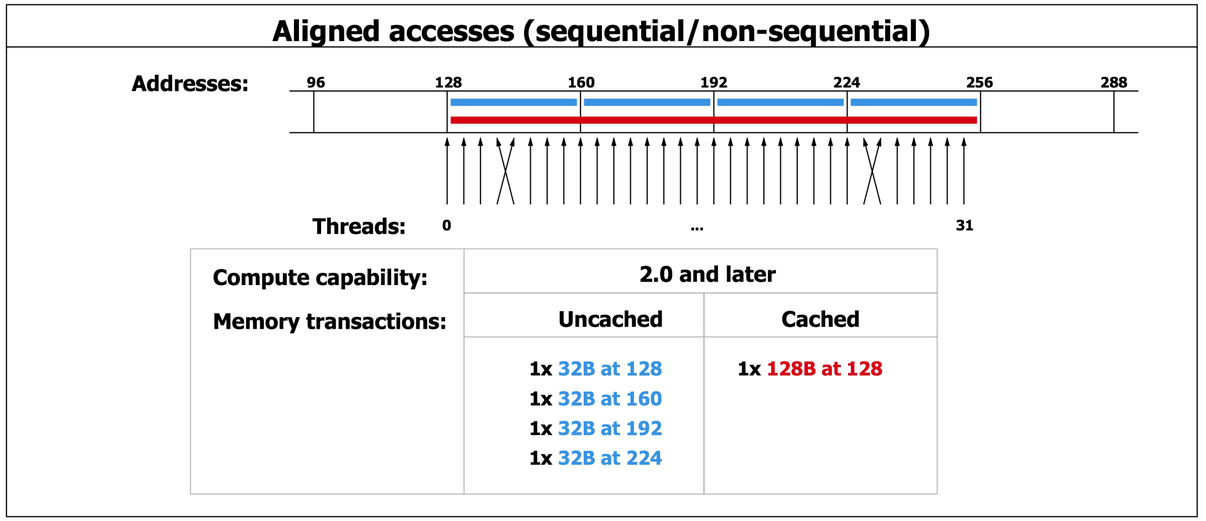
- L2 cache line: 32 bytes
- Memory segment: 128 bytes
- When memory accesses are aligned and coalesced, we get the maximum bandwidth.
- Bandwidth goes down if:
  - **Memory access is unaligned**
  - **Memory access has a stride**

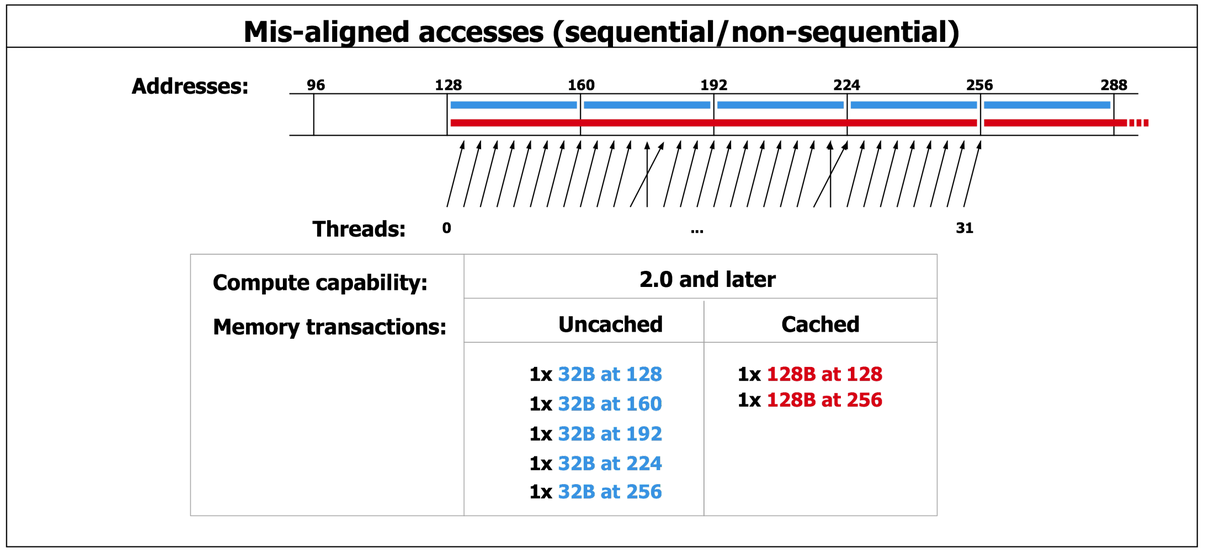
Examples of global memory accesses by a warp

4-byte word per thread

Uncached (blue) = L2

Cached (red) = L1



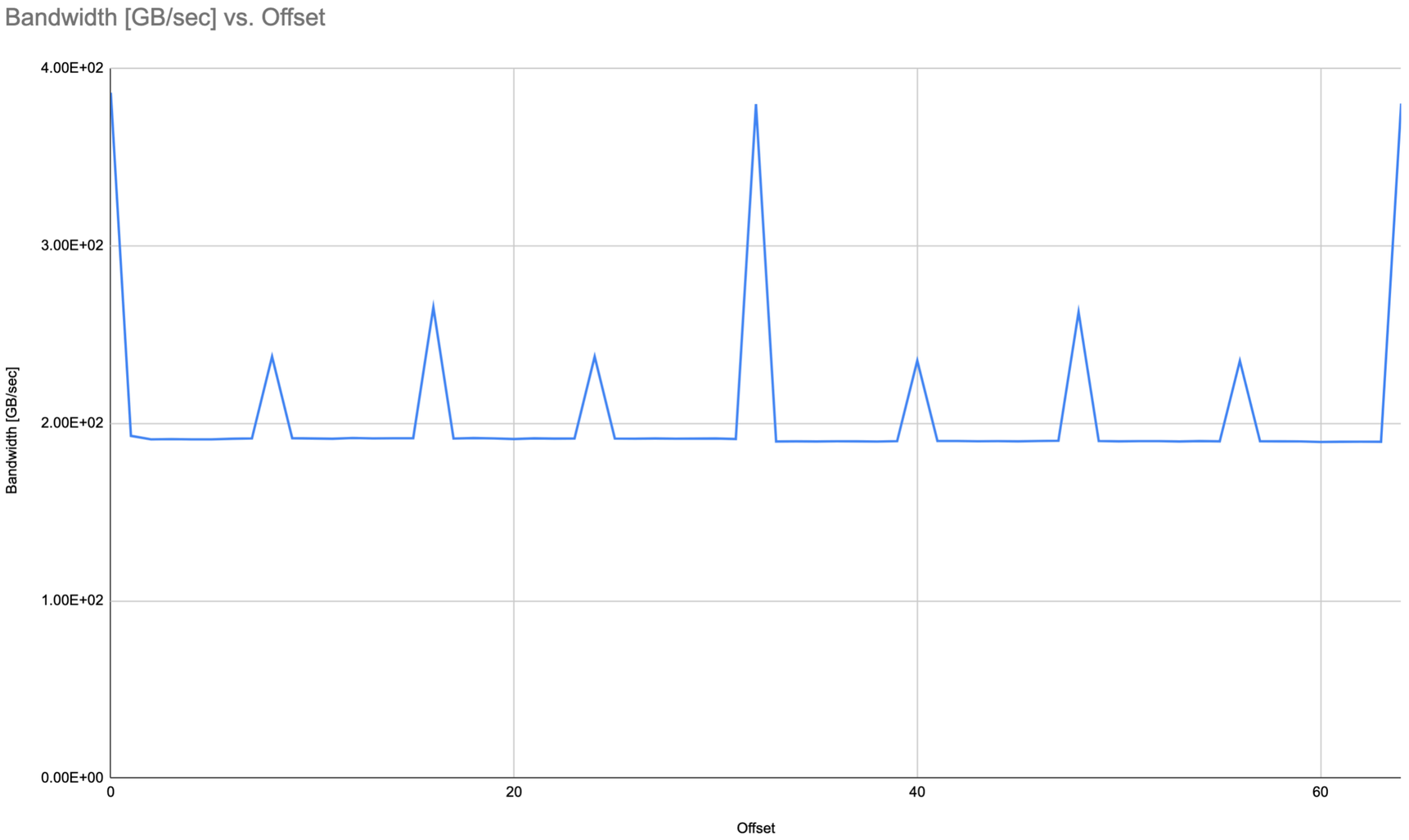


Offset benchmark

Let’s access memory but with an offset.

```
__global__ void offsetCopy(size_t n, float* odata, float* idata, int offset) {
  int wid = threadIdx.x / 32;
  size_t xid = blockIdx.x * 4 * blockDim.x + 4 * 32 * wid + threadIdx.x + offset;
  xid = xid % n;
  odata[xid] = idata[xid];
}
```

Bandwidth vs offset for float type

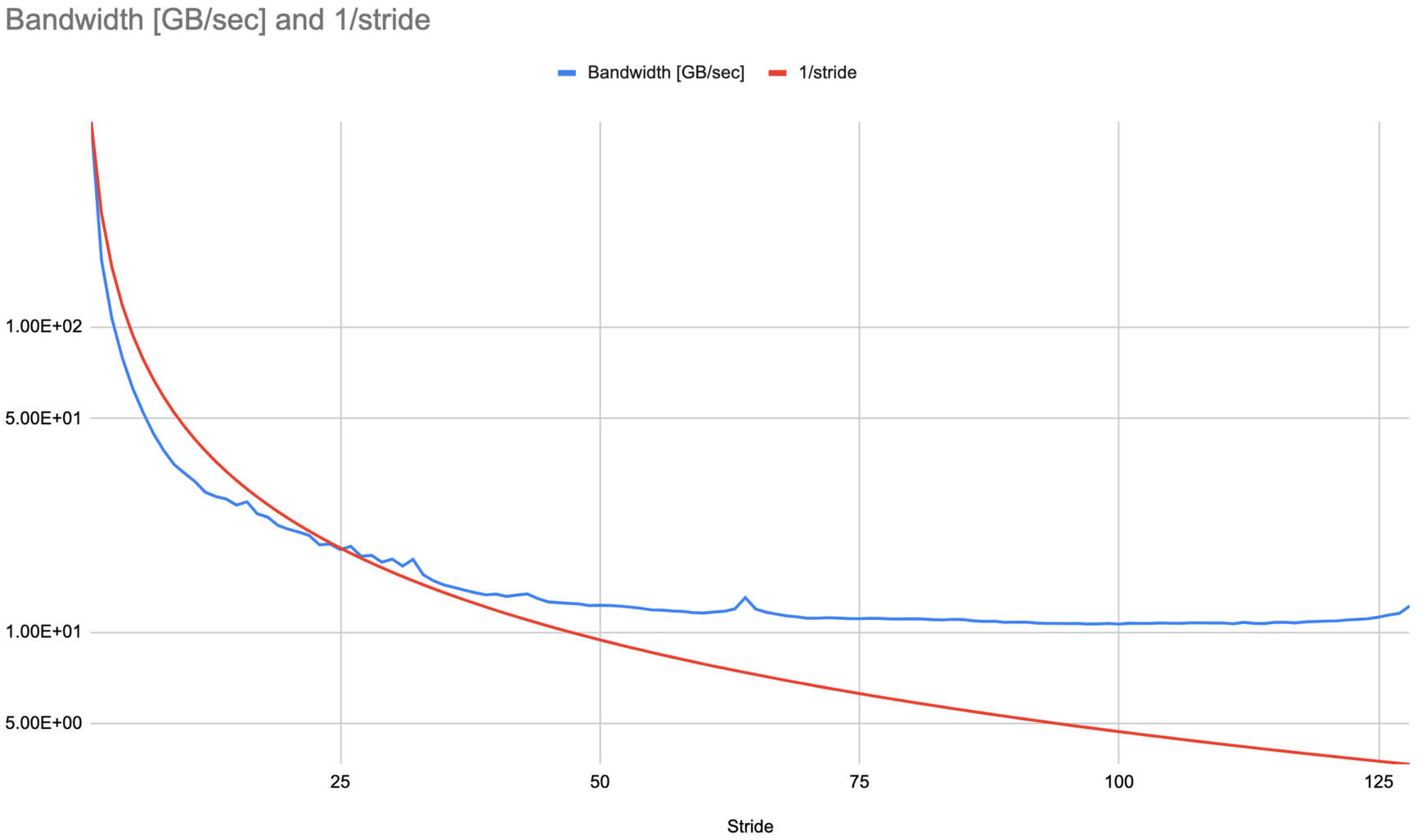


Strided access

Same code, but this time using a stride.

```
__global__ void stridedCopy(size_t n, float* odata, float* idata, int stride) {
  size_t xid = stride * (blockIdx.x * blockDim.x + threadIdx.x);
  xid = xid % n;
  odata[xid] = idata[xid];
}
```

Bandwidth vs stride for float type



Memory hierarchy

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

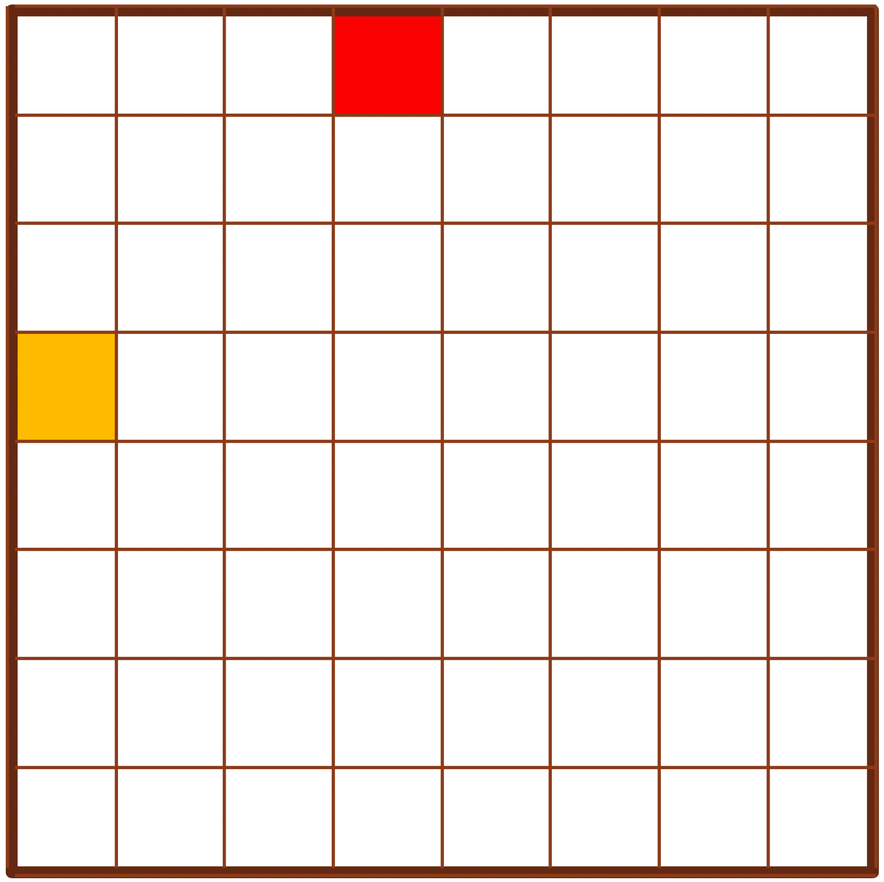
† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

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

Example application

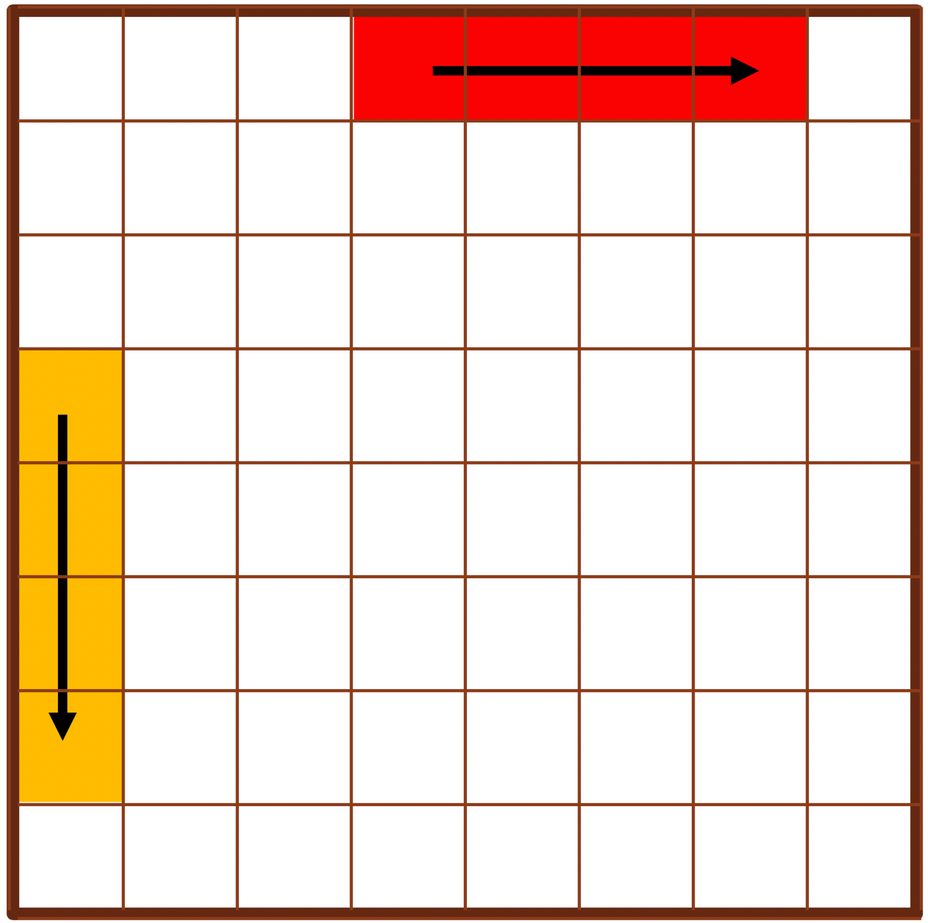
- Let’s put all these concepts into play through a specific example: a matrix transpose.
- It’s all about bandwidth!

Transposing a matrix



Algorithm 1

Assign a thread block to a single row/column



Code

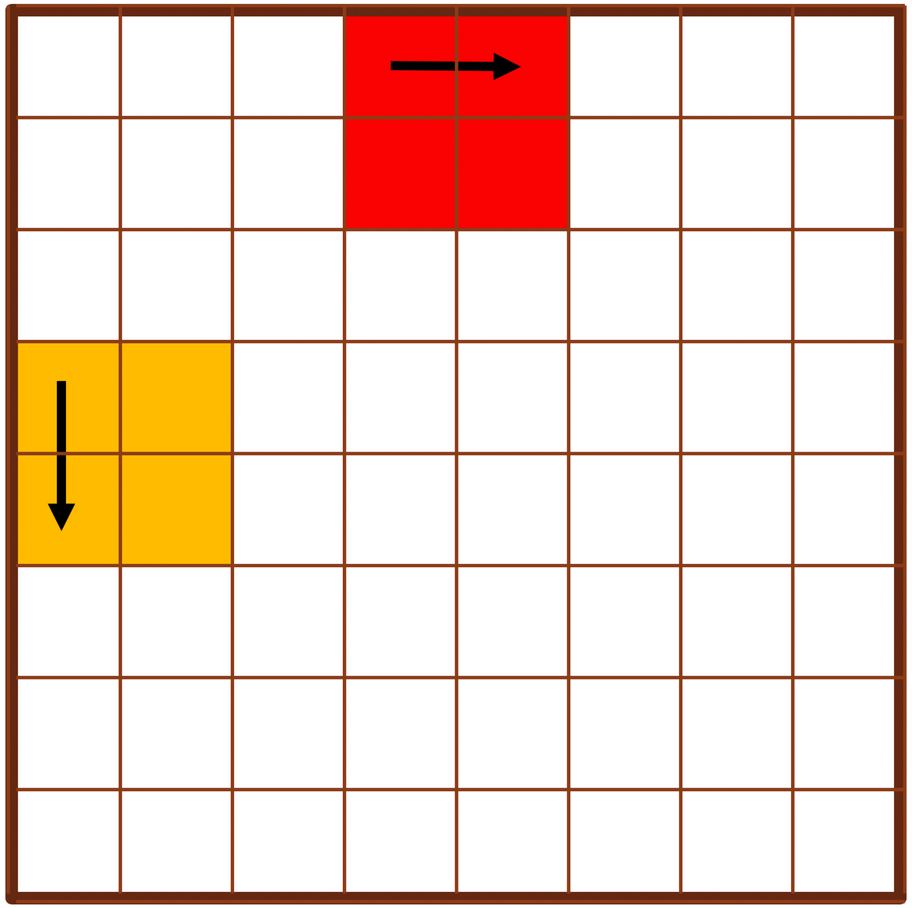
```
__global__ void simpleTranspose(int *array_in, int *array_out, size_t n_rows,
size_t n_cols)
{
  const size_t tid = threadIdx.x + blockDim.x * blockIdx.x;

  size_t col = tid % n_cols;
  size_t row = tid / n_cols;

  if (col < n_cols && row < n_rows)
  {
    array_out[col * n_rows + row] = array_in[row * n_cols + col];
  }
}
```

- Read: good
- Write: bad

Matrix block

- Instead, it is preferable if a thread block accesses a block.
- This results in a better use of the cache.
- Cache hits increase.

Block access



Code

```
__global__ void simpleTranspose2D(int *array_in, int *array_out, size_t n_rows,
size_t n_cols)
{
  const size_t col = threadIdx.x + blockDim.x * blockIdx.x;
  const size_t row = threadIdx.y + blockDim.y * blockIdx.y;

  if (col < n_cols && row < n_rows)
  {
    array_out[col * n_rows + row] = array_in[row * n_cols + col];
  }
}
```

A single thread block works with multiple rows and columns.

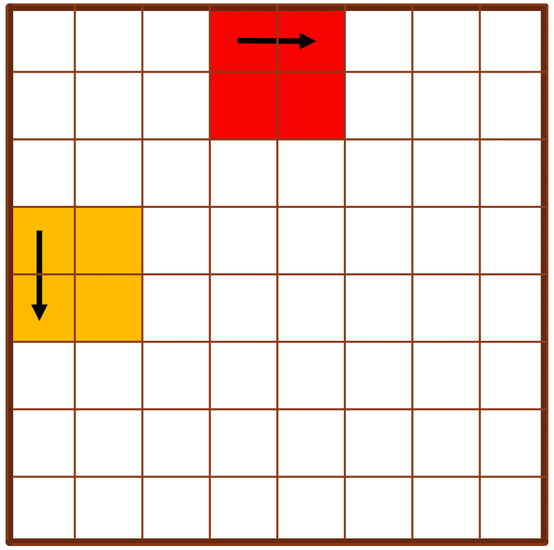
Kernel configuration

```
dim3 block_dim(8, 32);
dim3 grid_dim(n / 8, n / 32);
simpleTranspose2D<<<grid_dim, block_dim>>>(d_in, d_out, n, n);
```

256 threads in a block arranged as a 8 x 32 block.

Memory access for a warp

```
array_out[col * n_rows + row] = array_in[row * n_cols + col];
```



- For a given warp:
  - column: 0 to 7 (contiguous during read)
  - row: 0 to 3 (contiguous during write)
- Read: warp reads 32 (8x4) coalesced bytes
- Write: warp writes 16 (4x4) coalesced bytes

Benchmark results

```
[ RUN      ] TransposeFixture.memcopy
Number of MB to transpose: 4096

Bandwidth bench
GPU took 17.6967 ms
Effective bandwidth is 485.398 GB/s
[       OK ] TransposeFixture.memcopy (18016 ms)
[ RUN      ] TransposeFixture.simple
Number of MB to transpose: 4096

simpleTranspose
GPU took 286.038 ms
Effective bandwidth is 30.0308 GB/s
[       OK ] TransposeFixture.simple (49073 ms)
[ RUN      ] TransposeFixture.transpose2D
Number of MB to transpose: 4096

simpleTranspose2D
GPU took 27.6503 ms
Effective bandwidth is 310.664 GB/s
[       OK ] TransposeFixture.transpose2D (45568 ms)
```

Can we reconcile the reads and writes?

Shared memory

- Shared memory is very fast
- Bandwidth does not depend on stride (with some caveats)
- Idea:
  - Load a square block of data in shared memory
  - Perform the transpose in shared memory
  - Write back the result

Where is shared memory?

Characteristics of the shared memory

- Threads **within a block** can cooperate by sharing data through shared memory.
- For efficient cooperation, the shared memory is expected to be a **low-latency memory** near each processor core (much like an L1 cache).

How to declare shared memory

- Since shared memory is local to a block, it cannot be allocated in the main function. Instead, **it needs to be declared inside the CUDA kernel.**
- `lane` id of thread inside warp
- `block` variable allocated in shared memory

```
template <int num_warps>
__global__ void fastTranspose(int *array_in, int *array_out, size_t n_rows,
size_t n_cols)
{
  const int warp_id = threadIdx.y;
  const int lane = threadIdx.x;

  __shared__ int block[warp_size][warp_size];
```

Load the data into shared memory

```
// Load 32x32 block into shared memory
size_t gc = bc * warp_size + lane; // Global column index
size_t gr;
for (int i = 0; i < warp_size / num_warps; ++i)
{
  gr = br * warp_size + i * num_warps + warp_id; // Global row index
  block[i * num_warps + warp_id][lane] = array_in[gr * n_cols + gc];
}
__syncthreads();
```

- Reads follow a coalesced pattern.
- `__syncthreads` : all threads in the block need to wait and synchronize at that instruction.
- This ensures that all the data has been loaded in block before proceeding.

Write back the data

```
gr = br * warp_size + lane;
for (int i = 0; i < warp_size / num_warps; ++i)
{
  gc = bc * warp_size + i * num_warps + warp_id;
  array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
}
```

- Writes are also coalesced.
- We should now see a significant increase in speed!

Instead

```
[ RUN      ] TransposeFixture.memcopy
Number of MB to transpose: 4096

Bandwidth bench
GPU took 18.1248 ms
Effective bandwidth is 473.932 GB/s
[       OK ] TransposeFixture.memcopy (18004 ms)
[ RUN      ] TransposeFixture.simple
Number of MB to transpose: 4096

simpleTranspose
GPU took 286.019 ms
Effective bandwidth is 30.0328 GB/s
[       OK ] TransposeFixture.simple (49049 ms)
[ RUN      ] TransposeFixture.transpose2D
Number of MB to transpose: 4096

simpleTranspose2D
GPU took 27.6556 ms
Effective bandwidth is 310.604 GB/s
[       OK ] TransposeFixture.transpose2D (47147 ms)
[ RUN      ] TransposeFixture.fastTranspose
Number of MB to transpose: 4096

fastTranspose
GPU took 24.942 ms
Effective bandwidth is 344.396 GB/s
[       OK ] TransposeFixture.fastTranspose (44844 ms)
```

- Performance has increased.
- But more can be done…

Shared memory has some constraints

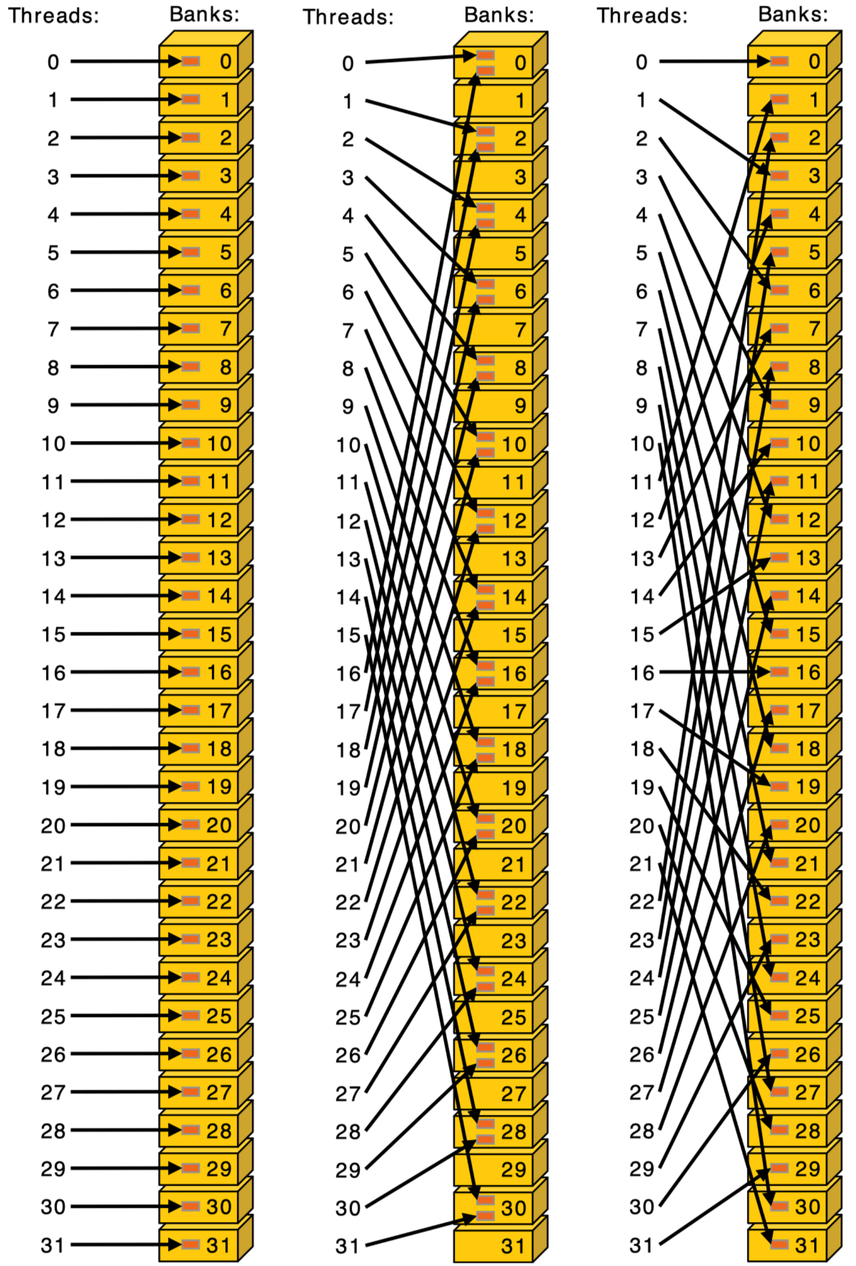
- In order to be fast, the shared memory needs to be parallel.
- To achieve this, the shared memory uses **32 banks** capable of copying data to and from the register files.
- Banks can be accessed simultaneously, achieving peak bandwidth.
- However, if 2 threads request different words in the same bank, the process becomes **sequential and the bandwidth drops.**
- This is a called a **bank conflict.**

The rules of shared memory

- Shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously.
- Any memory read or write request made of n addresses that fall in n distinct memory banks can be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.
- However, if two addresses of a memory request fall in the same memory bank, there is a **bank conflict** and the access has to be **serialized.**
- The hardware **splits a memory request** with bank conflicts into as many separate **conflict-free requests** as necessary, **decreasing throughput by a factor equal to the number of separate memory requests.**
- If the number of separate memory requests is n, the initial memory request is said to cause **n-way bank conflicts.**

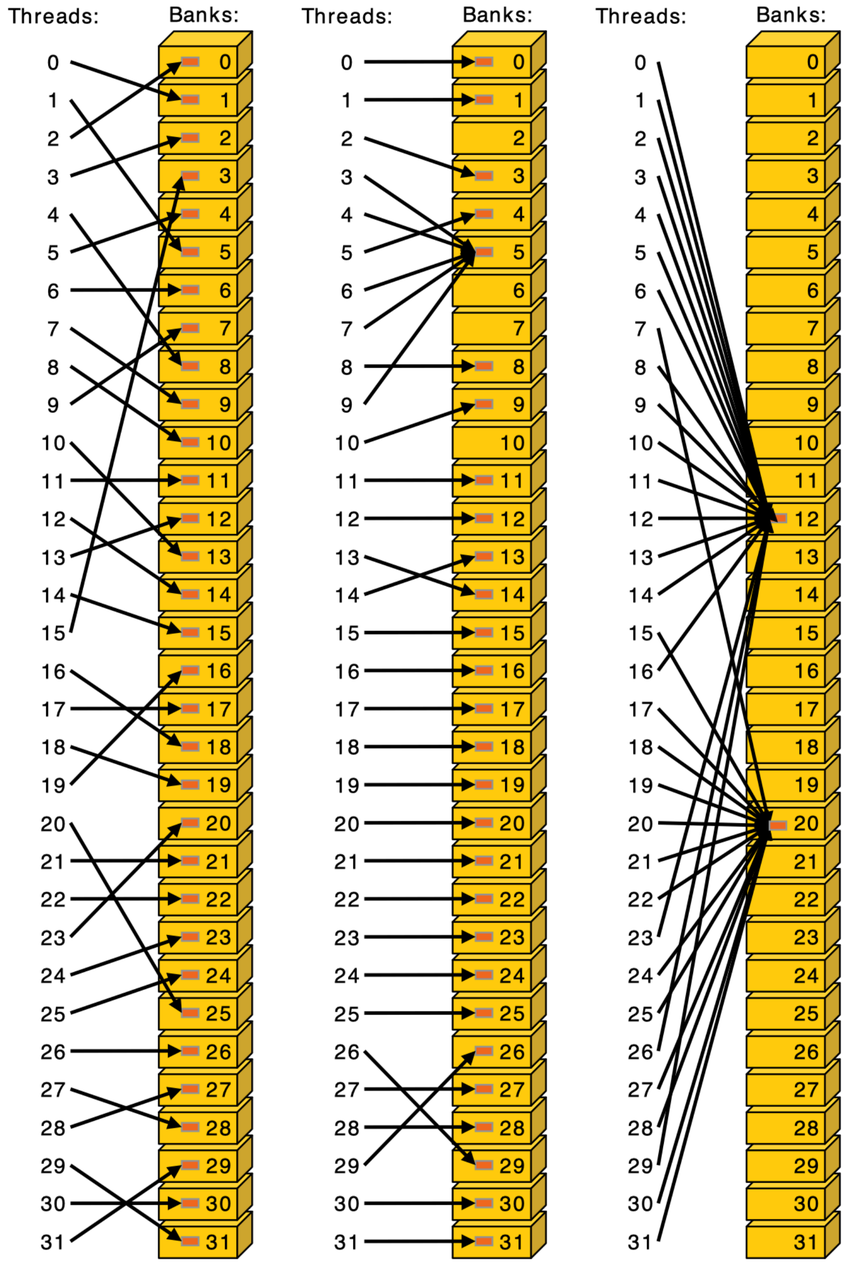
Strided access

- Strided shared memory access.
- Left: no bank conflict
- Middle: two-way bank conflict with stride 2
- Right: the stride is odd; no bank conflicts are present.



Irregular access

- Left: random permutation; no bank conflict
- Middle: conflict-free access on 3, 4, 6, 7, and 9 since they access the same word in bank 5.
- Right: conflict-free **broadcast** access; all threads access **the same word** within a bank.



What strided access do we have for the shared memory?

```
// Load 32x32 block into shared memory
size_t gc = bc * warp_size + lane; // Global column index
size_t gr;
for (int i = 0; i < warp_size / num_warps; ++i)
{
  gr = br * warp_size + i * num_warps + warp_id; // Global row index
  block[i * num_warps + warp_id][lane] = array_in[gr * n_cols + gc];
}
__syncthreads();
```

Stride is 1

What happens during the write?

```
gr = br * warp_size + lane;
for (int i = 0; i < warp_size / num_warps; ++i)
{
  gc = bc * warp_size + i * num_warps + warp_id;
  array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
}
```

Stride is 32!

There is a simple fix

```
__shared__ int block[warp_size][warp_size];
```

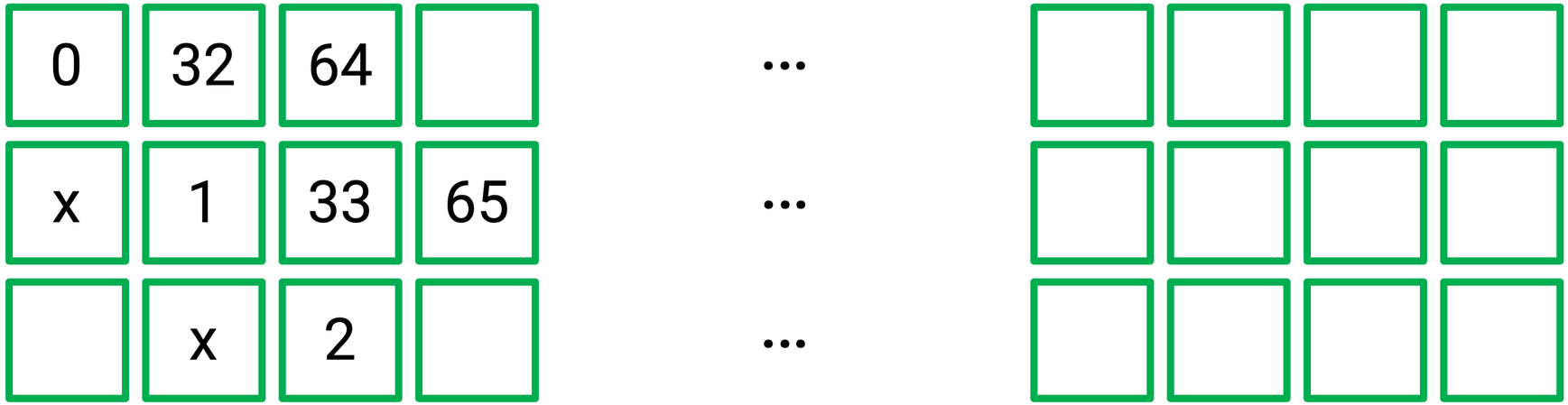
```
__shared__ int block[warp_size][warp_size+1];
```

- We make the stride odd by adding 1 to the number of columns.
- Now the stride is 33 instead of 32.
- We avoid all bank conflicts!

What happens

```
__shared__ int block[warp_size][warp_size+1];
```

```
array_out[gc * n_rows + gr] = block[lane][i * num_warps + warp_id];
```



New performance

Although we have not yet reached the peak bandwidth, we are doing better than `simpleTranspose2D`.

```
[ RUN      ] TransposeFixture.memcopy
Number of MB to transpose: 4096

Bandwidth bench
GPU took 17.6967 ms
Effective bandwidth is 485.398 GB/s
[       OK ] TransposeFixture.memcopy (18016 ms)
[ RUN      ] TransposeFixture.simple
Number of MB to transpose: 4096

simpleTranspose
GPU took 286.038 ms
Effective bandwidth is 30.0308 GB/s
[       OK ] TransposeFixture.simple (49073 ms)
[ RUN      ] TransposeFixture.transpose2D
Number of MB to transpose: 4096

simpleTranspose2D
GPU took 27.6503 ms
Effective bandwidth is 310.664 GB/s
[       OK ] TransposeFixture.transpose2D (45568 ms)
[ RUN      ] TransposeFixture.fastTranspose
Number of MB to transpose: 4096

fastTranspose
GPU took 24.539 ms
Effective bandwidth is 350.052 GB/s
[       OK ] TransposeFixture.fastTranspose (45321 ms)
```