CSCI-GA.3033-004

Graphics Processing Units (GPUs): Architecture and Programming

CUDA
Advanced Techniques 1

Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

http://www.mzahran.com

Some Refreshing Exercises!

- Suppose registers and shared memory capacities were not an issue. When is it still beneficial to put values fetched from memory into the shared memory?

Some Refreshing Exercises!

- Assume a kernel is launched with 1000 blocks. Each block has 512 threads.
  - If a variable is declared as local in the kernel, how many versions will be created throughout the lifetime of the kernel?
 - How about if the variable is created as shared?

Some Refreshing Exercises!

- A kernel contains 36 floating point operations and 7 32-bit word global memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute- or memory-bound.

    – Peak FLOPS = 200 GFLOPS, Peak Memory Bandwidth = 100 GB/s

    – Peak FLOPS = 300 GFLOPS, Peak Memory Bandwidth = 250 GB/s

A Note About Performance

Defining Performance

- Which airplane has the best performance?
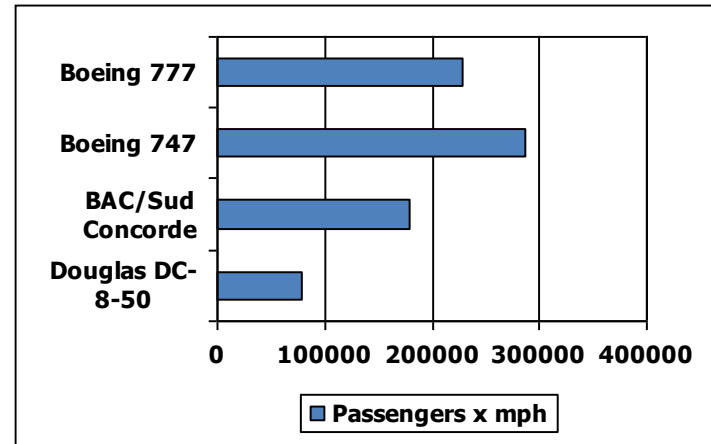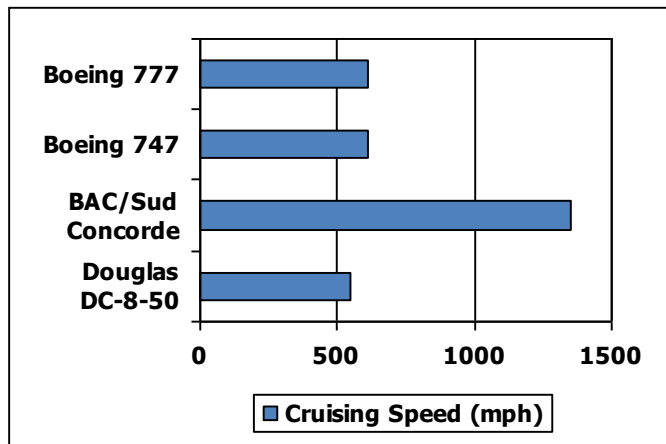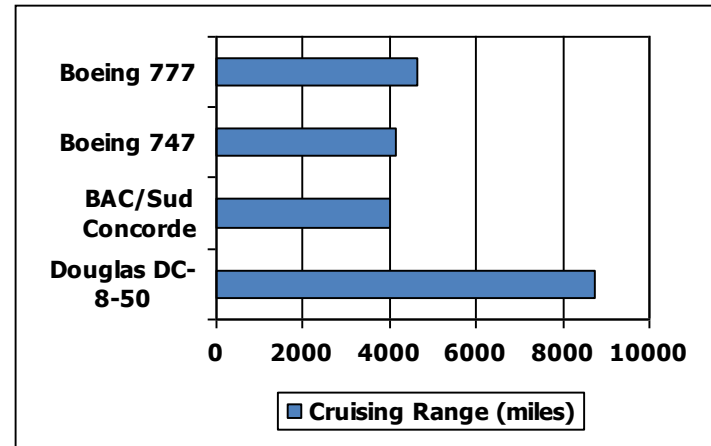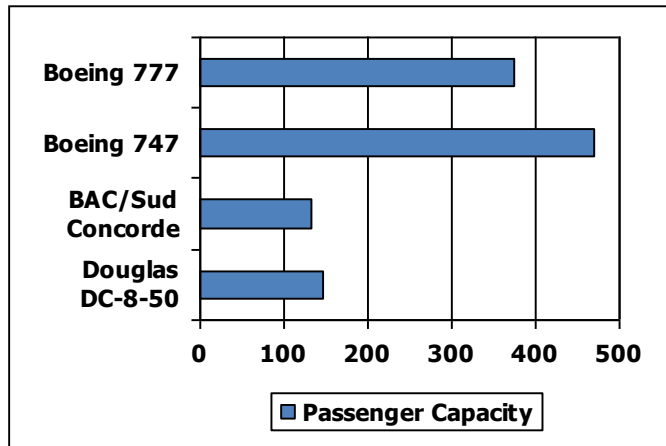
Performance Considerations

- There are many hardware constraints.
- Depending on the application, different constraints may dominate.
- We can improve performance of an application by trading one resource usage for another.

Performance Issue: Thread Diversion

- Works well when all threads in a warp follow the same control-flow
- Performance loss due to thread diversion
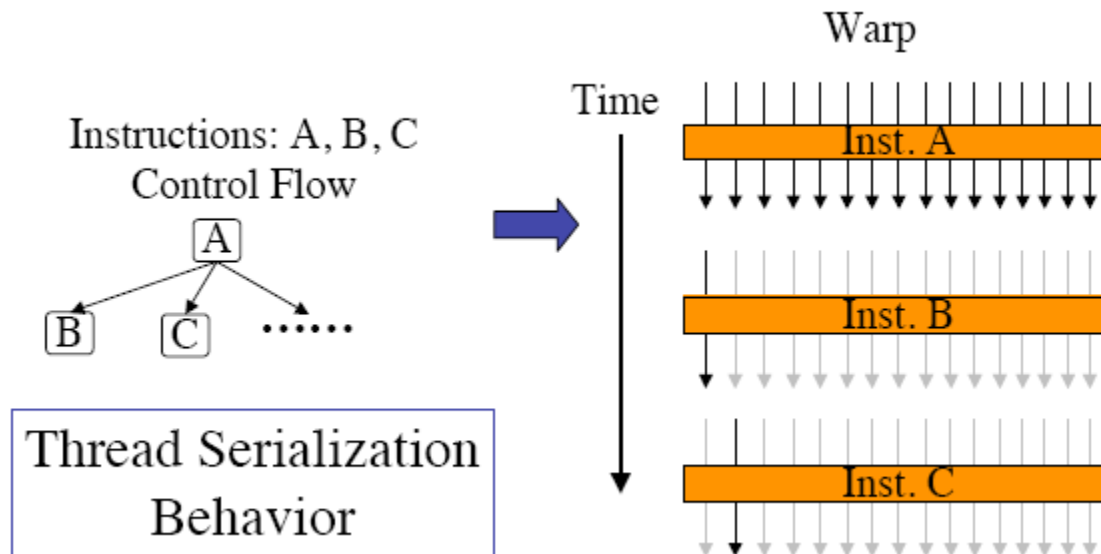
Performance Issue: Thread Diversion

```
1. __shared__ float partialSum[]

2. unsigned int t = threadIdx.x;
3. for (unsigned int stride = 1;
4.      stride < blockDim.x; stride *= 2)
5. {
6.     syncthreads();
7.  if (t % (2*stride) == 0)
8.    partialSum[t] += partialSum[t+stride];
9. }
```


Example: Sum Reduction Kernel

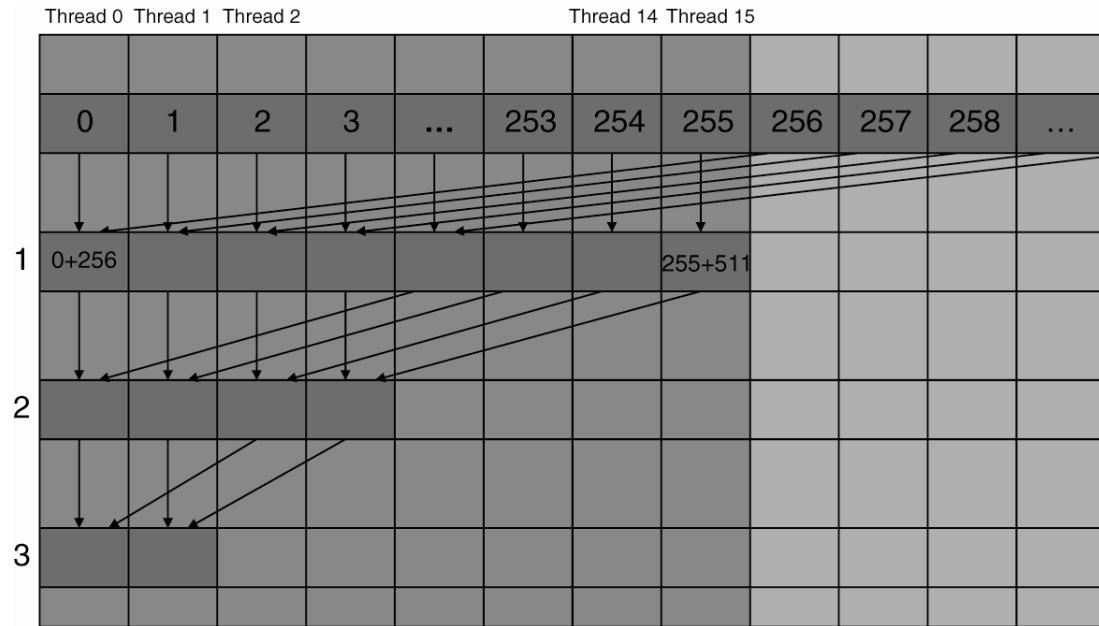# Performance Issue: Thread Diversion

```
1. __shared__ float partialSum[];

2. ursigned int t = threadIdx.x;
3. for (unsigned int stride = blockDim.x >>1;
4.   stride > 0; stride >>= 1)
5. {
6.   __syncthreads();
7.   if (t < stride)
8.     partialSum[t] += partialSum[t+stride];
9. }
```

Why is this version better than the previous one?

Example: Sum Reduction Kernel

Performance Issue: Global Memory

- Typical application: process massive amount of data within short period of time
  - From global memory
  - large amount + short period = huge bandwidth requirement

- Two main challenges regarding global memory:
  - Long latency
  - Relatively limited bandwidth

Dealing With Global Memory: TILING

- We have seen this before
- Make use of shared memory available in SMs to reduce trips to global memory

Dealing With Global Memory: Coalescing

- To more effectively move data from global memory to shared memory and registers

- For best results: can be used with tiling

- Global memory:
  - DRAM
  - Reading a bit is slow
  - So memory is implemented to read several bits in parallel

Dealing With Global Memory: Coalescing

- If an application can make use of data from multiple consecutive locations, the DRAM can supply the data in much higher rate.

- Kernel must arrange its data access accordingly

- When all threads in a warp execute a load instruction:

  – The hardware detects whether the addresses are consecutive

  – The hardware combines (coalesces) all accesses in a consolidated access to consecutive DRAM locations

Dealing With Global Memory: Coalescing

Not coalesced   coalesced

Same warp { Thread 1 / Thread 2

WIDTH
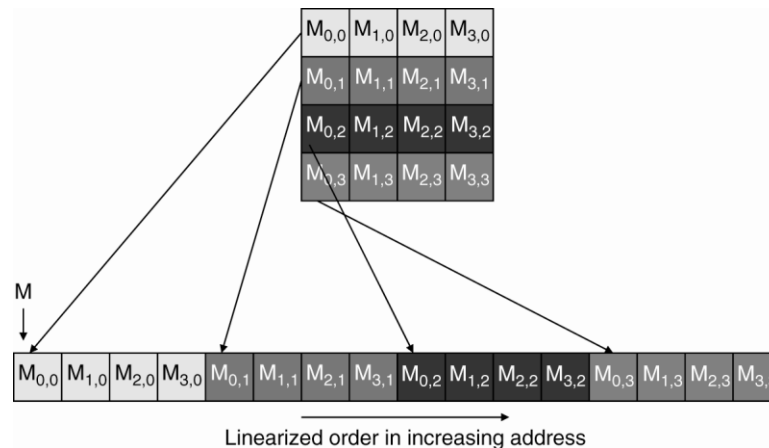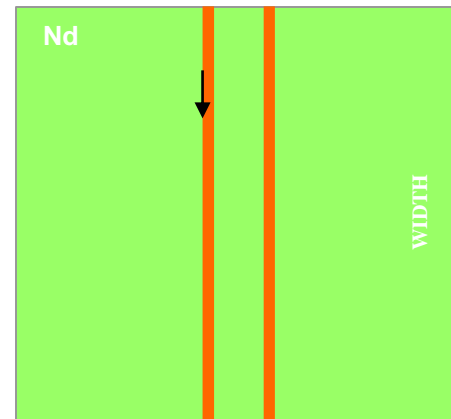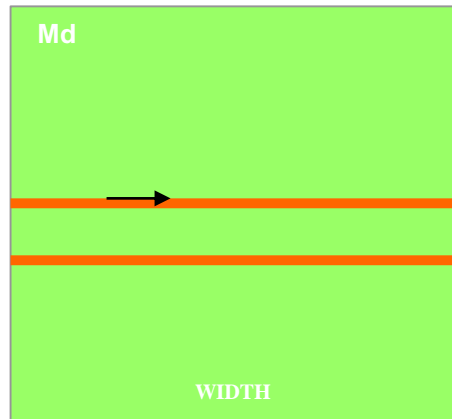
Dealing With Global Memory: Coalescing

Favorable access
direction in
Kernel code
of one thread

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

Time Period 1				Time Period 2				...
T_0	T_1	T_2	T_3	T_0	T_1	T_2	T_3	

M

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

Dealing With Global Memory: Coalescing

Access direction in Kernel code

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

...

Time Period 2

T_1 T_2 T_3 T_4

Time Period 1

T_1 T_2 T_3 T_4

M

| $M_{0,0}$ | $M_{1,0}$ | $M_{2,0}$ | $M_{3,0}$ | $M_{0,1}$ | $M_{1,1}$ | $M_{2,1}$ | $M_{3,1}$ | $M_{0,2}$ | $M_{1,2}$ | $M_{2,2}$ | $M_{3,2}$ | $M_{0,3}$ | $M_{1,3}$ | $M_{2,3}$ | $M_{3,3}$ |
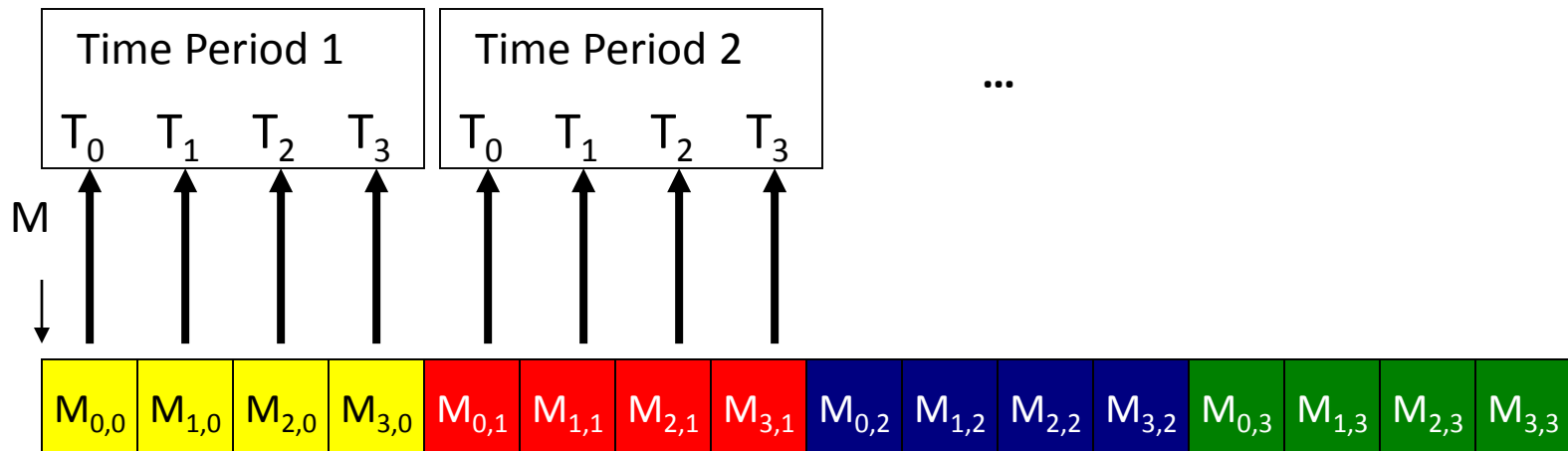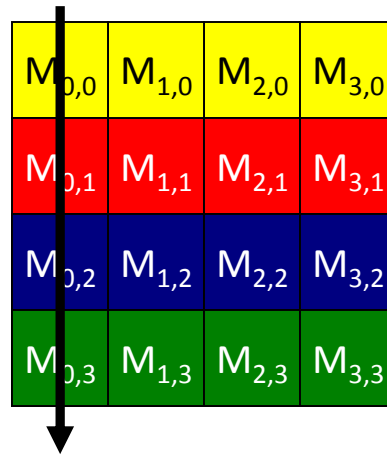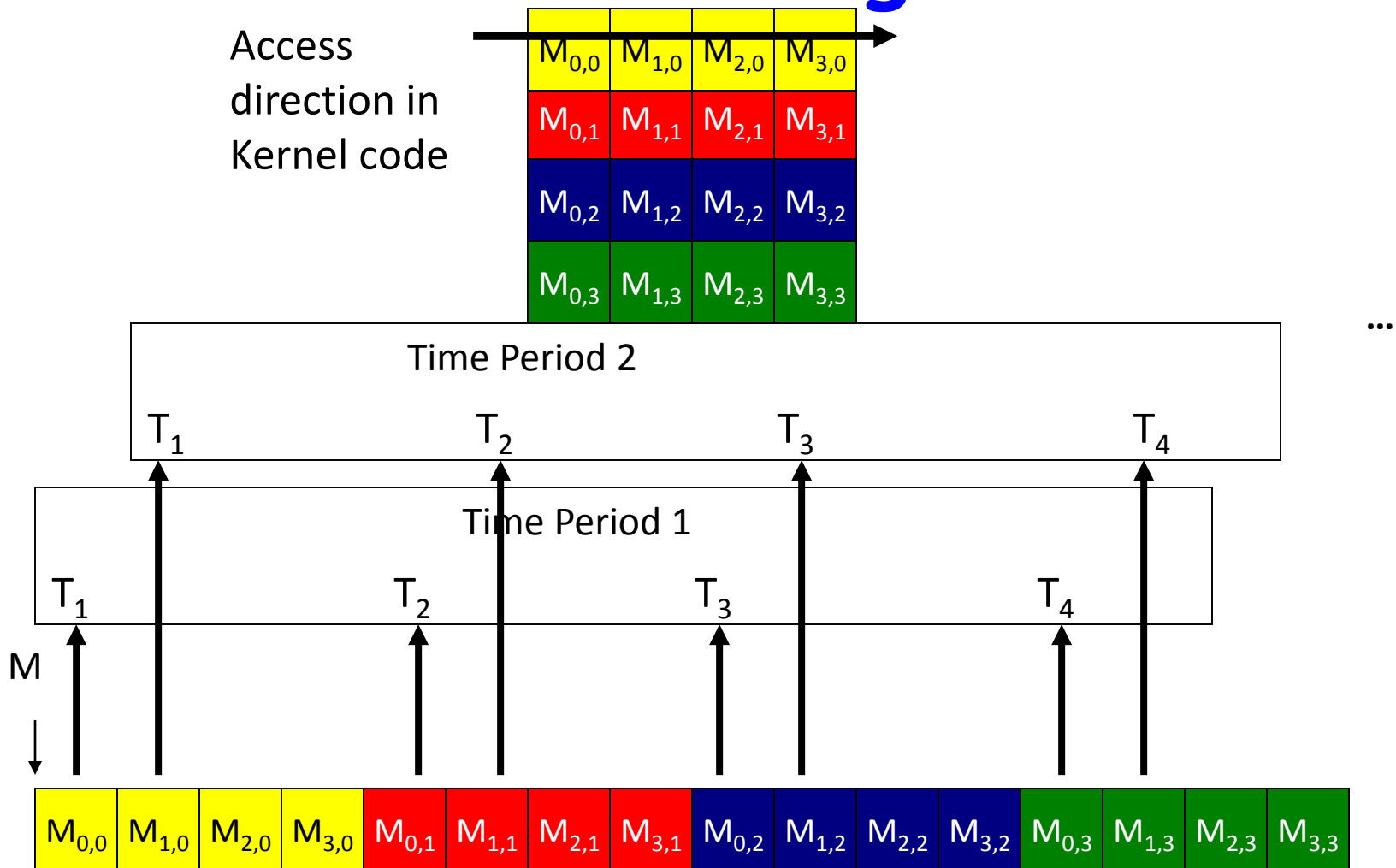
Dealing With Global Memory: Cache

- Fermi (and later GPUs) have cache for global memory

- Caches automatically coalesce most of kernel access patterns
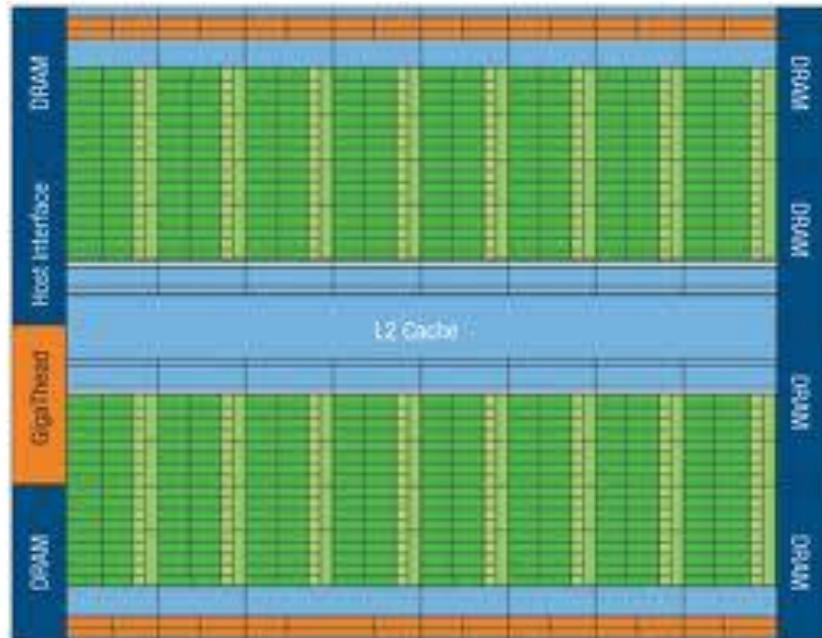
Dealing With Global Memory: Prefetching

Prefetch next data elements while consuming the current data elements.

This increases the number of independent instructions between memory accesses and consumers

```
Loop {

Load current tile to shared
    memory


__syncthreads()


Compute current tile


__syncthreads()
}
```
A   Without prefetching

```
Load first tile from global memory into
    registers

Loop {
Deposit tile from registers to shared
    memory
__syncthreads()

Load next tile from global memory into
    registers

Compute current tile


__syncthreads()
}
```
B   With prefetching

Performance Issue: SM Resources

- Execution resources in SM include:
  - registers
  - block slots
  - thread slots
- There is an interaction among the resources that you must take into account.

Performance Issue: SM Resources

Example: Assume G80 executing the matrix multiplication with 16x16 thread blocks
(8 block slots, 768 thread slots, 8192 registers)

If a thread needs 10 registers then:
- A block needs 10x16x16 = 2560 registers
- 3 blocks -> 7680 registers (under the 8192 limit)
- We can't add another block (will make it 10240)
- 3 blocks x 256 threads/block = 768 (within limit)

By using 1 extra variable the program saw a 1/3 reduction in warp parallelism **-> performance cliff**

Assume the programmer declares one more auto var:
- 11 x 16 x 16 = 2816 registers per block
- 3 blocks -> 3 x 2816 = 8448 (above limit)
- SM reduces #blocks by 1 -> 5632 registers required
- This reduces the number of threads in SM to 2 x256 -> 512

Performance Issue: SM Resources

Example: Still with G80:
- An instruction takes 4 cycles
- Assume 4 independent instructions between global memory load and its use
- Global memory latency is 200 cycles

To keep execution units fully utilized:

We need to have 200/(4 x 4) = 14 warps

Assume an extra register allows the programmer to use a transformation to increase independent instructions from 4 to 8, then:
- Now we need 200/(4 x 8) = 7 warps
- Blocks reduced from 3 to 2 -> warps reduced from 24 to 16
- Still we can fully utilize execution units

Trading thread-level parallelism with increased thread performance

Performance Issue: Instruction Mix

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];
```

Is the above code efficient?

- Extra instructions to update loop counter
- Extra instructions for conditional branch at the end of each iteration
- Using k to access matrices incurs address arithmetic instructions.
- All of the above compete with the floating-point calculations for limited instruction processing bandwidth.

2 FP arithmetic
2 address arithmetic instructions
1 loop branch instructions
1 loop increment instructions

only 1/3 instructions are FP operations

Performance Issue: Instruction Mix

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];
```
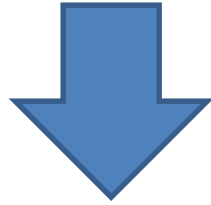


```
Pvalue += Ms[ty][0] * Ns[0][tx] + …
          Ms[ty][15] * Ns[15][tx];
```

Loop unrolling

Performance Issue: Thread Granularity

- Algorithmic decision
- It is often advantageous to put more work into each thread and use fewer threads
  - When redundant work exists between threads
  - Example: Let a thread compute 2 tiles
+ Less redundant work
+ Potentially more independent instructions
- More resources requirements

Putting It All Together
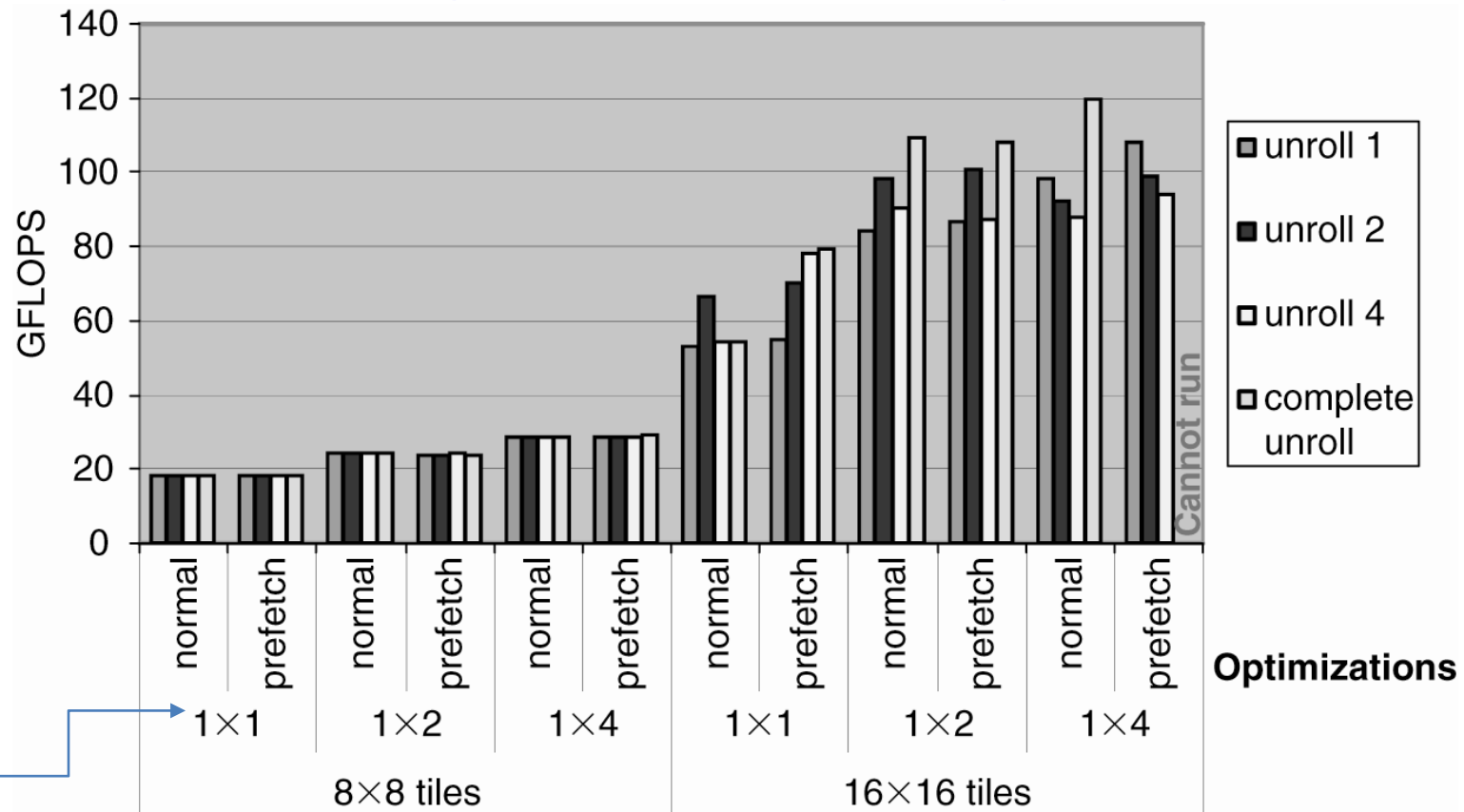
Thread Granularity:

- normal
- merging 2 blocks
- merging 4 blocks

Putting It All Together
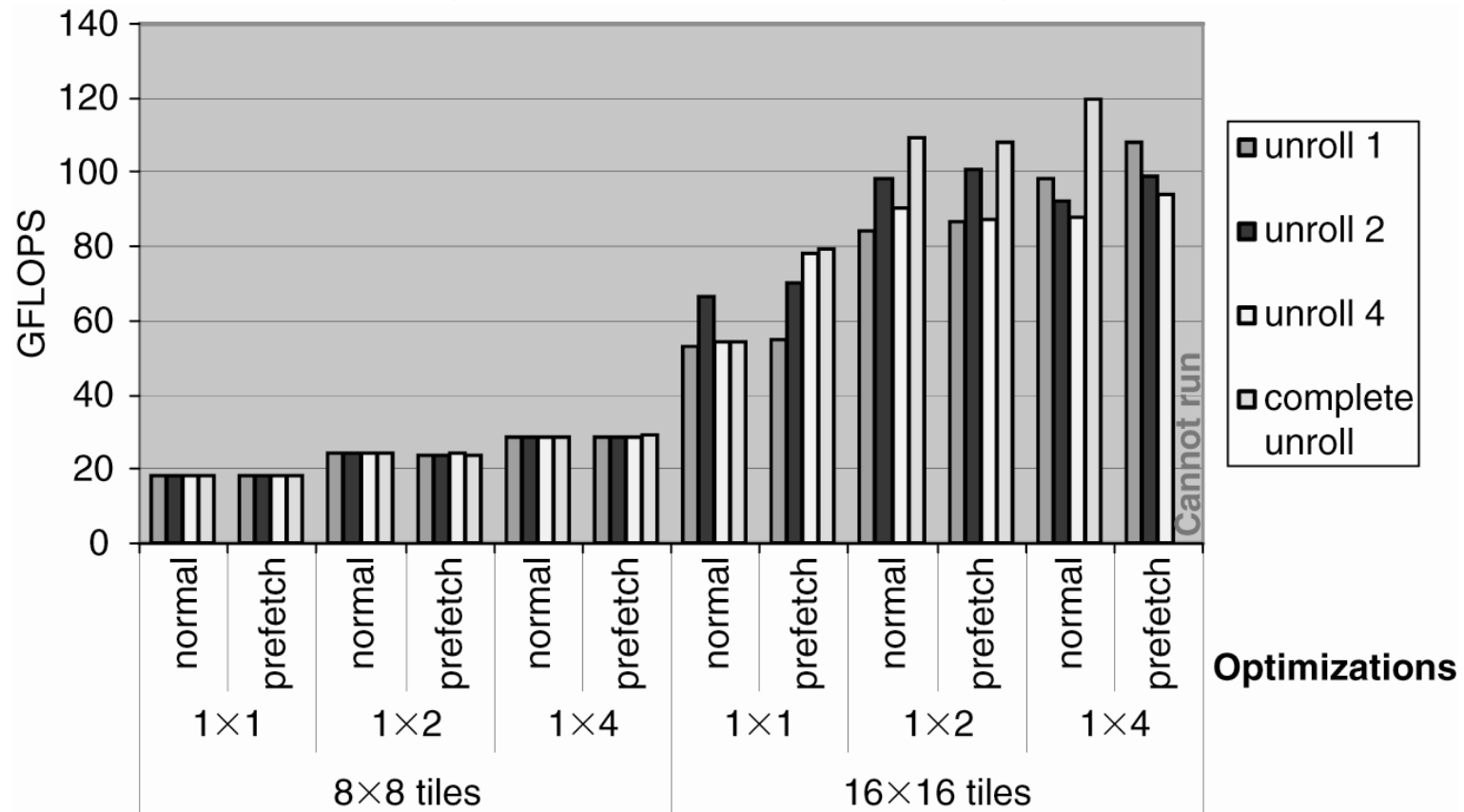
Until tile reaches 16x16 neither loop unrolling nor data prefetch helps.

For small tile size, global memory bandwidth severely limits performance.

Putting It All Together
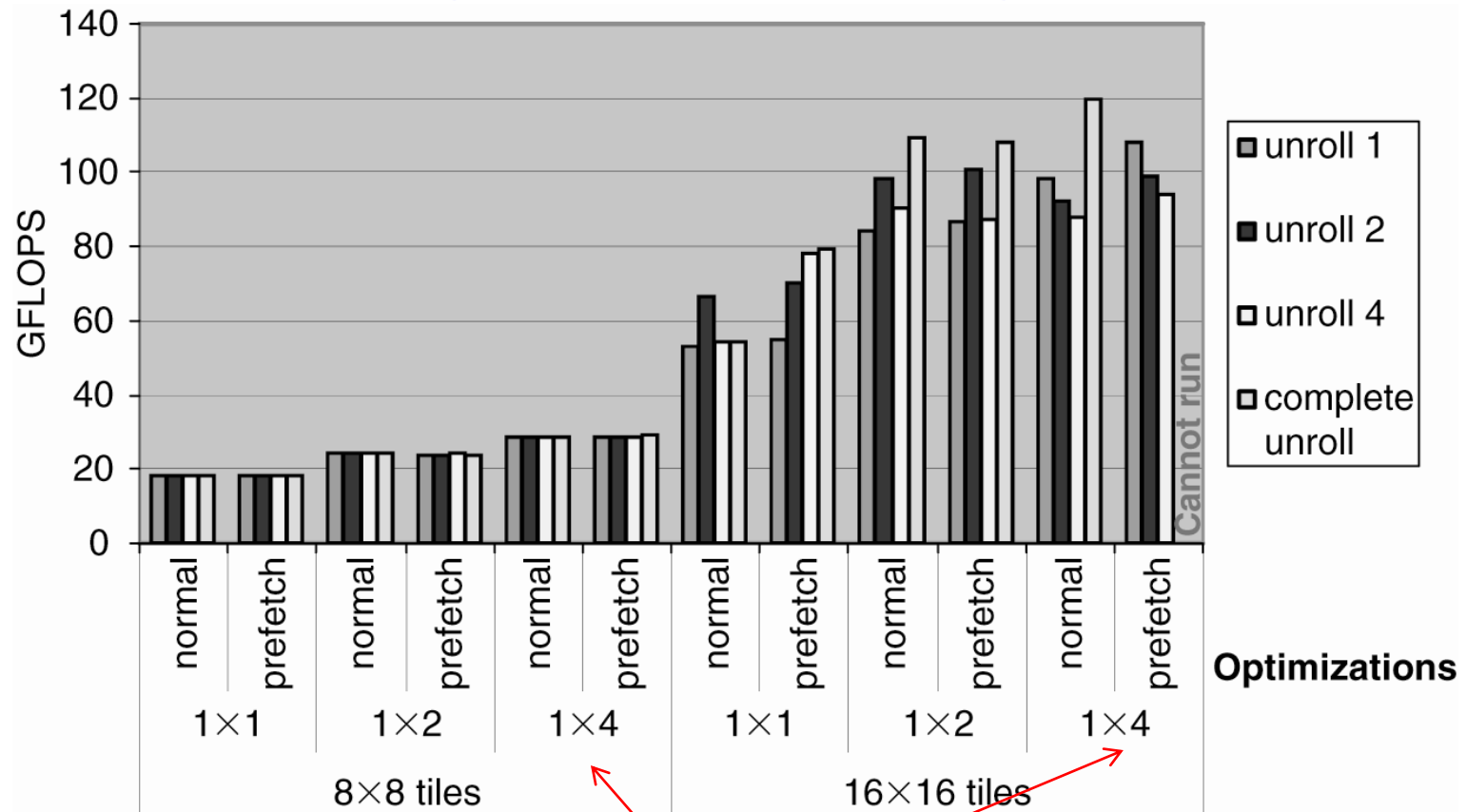
Granularity adjustment can reduce global memory access.

Putting It All Together
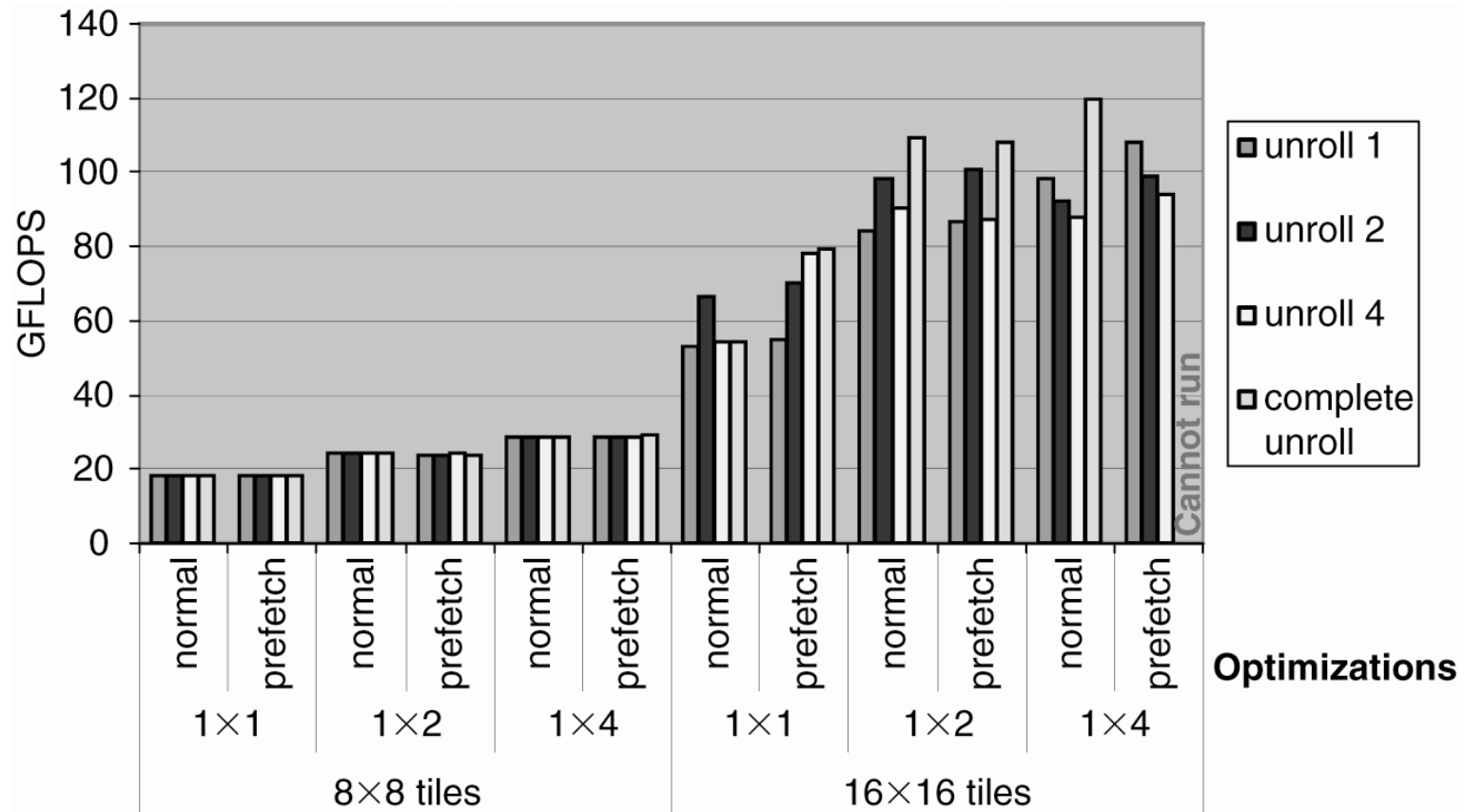
Data prefetching becomes less beneficial as as thread granularity increases.

About Algorithms for GPUs

- When designing an algorithm for GPU, the two main characteristics that determine its performance are:
  1. How much parallelism is available
  2. How much data must move through the memory hierarchy
- Then when you move from algorithm to code you need to take hardware constraints into account.

A note about compilation …
And some useful tools!

NVCC device specific switches

- `-arch` : controls the "virtual'' architecture that will be used for the generation of the PTX code.
- `-code` : specifies the actual device that will be targeted by the cubin binary.

Virtual Architecture (-arch)	Streaming Multipr. code (-code)	Feature Enabled
compute_10	sm_10	Basic features
compute_11	sm_11	global memory atomic operations
compute_12	sm_12	shared memory atomic operations and vote instructions
compute_13	sm_13	double precision floating point support
compute_20	sm_20	Fermi support
	sm_21	SM structure changes (e.g more cores)
compute_30	sm_30	Kepler support
compute_35	sm_35	Dynamic parallelism (enables recursion)
compute_50	sm_50	Maxwell support

Source: ***Multicore and GPU Programming: An Integrated Approach*** *by G. Barlas*

sm_xy

- x is the GPU generation number
- y is the version within that generation
- Binary compatibility of GPU applications is not guaranteed across different generations.
  - Example: a CUDA application that has been compiled for a Fermi GPU will very likely not run on a Kepler GPU (and vice versa).
- This is why nvcc relies on a two stage compilation model for ensuring application compatibility with future GPU generations.
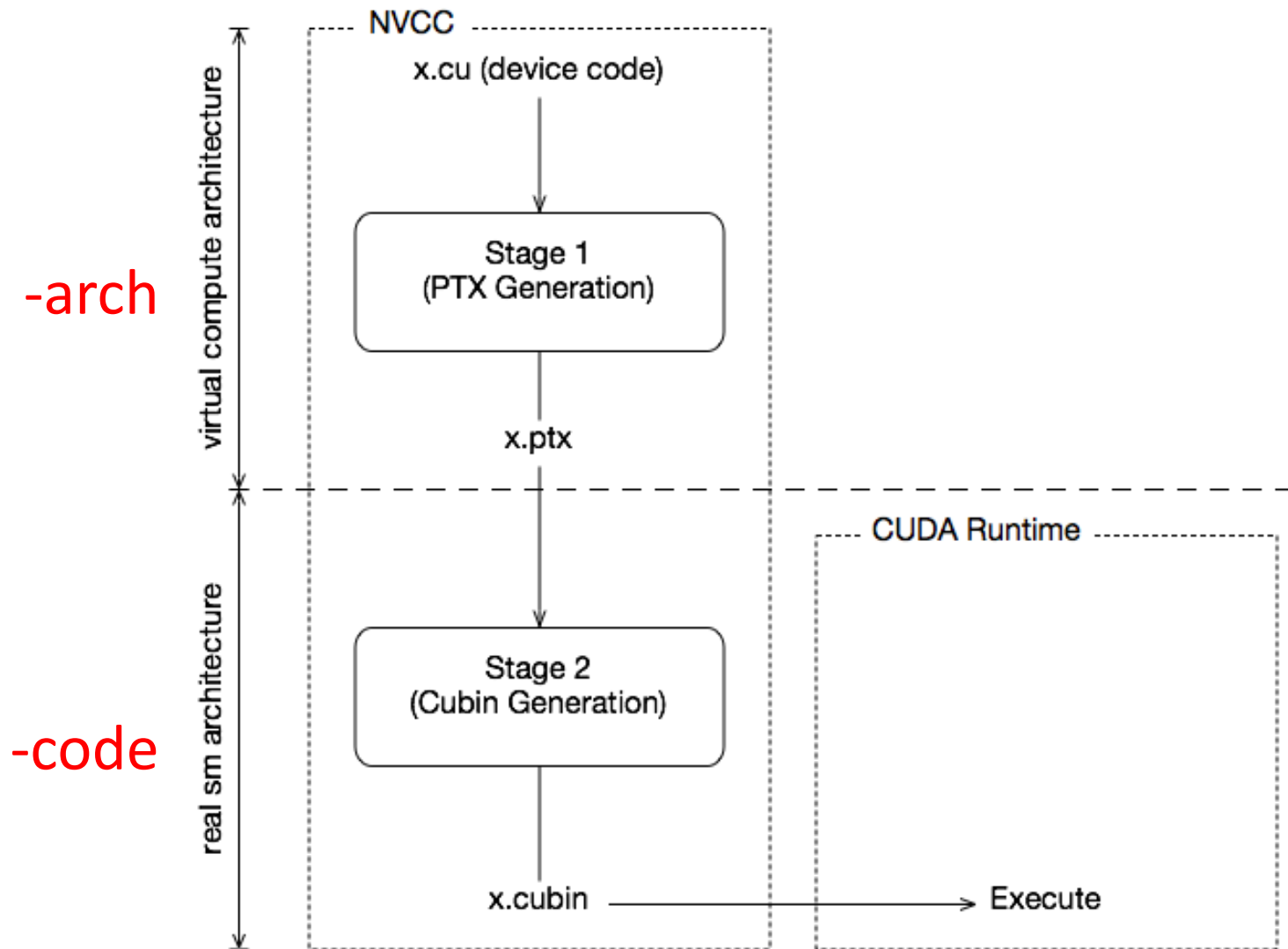
Source: CUDA compiler driver nvcc manual (NVIDIA website)
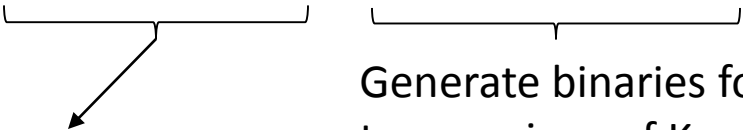
JIT Compilation
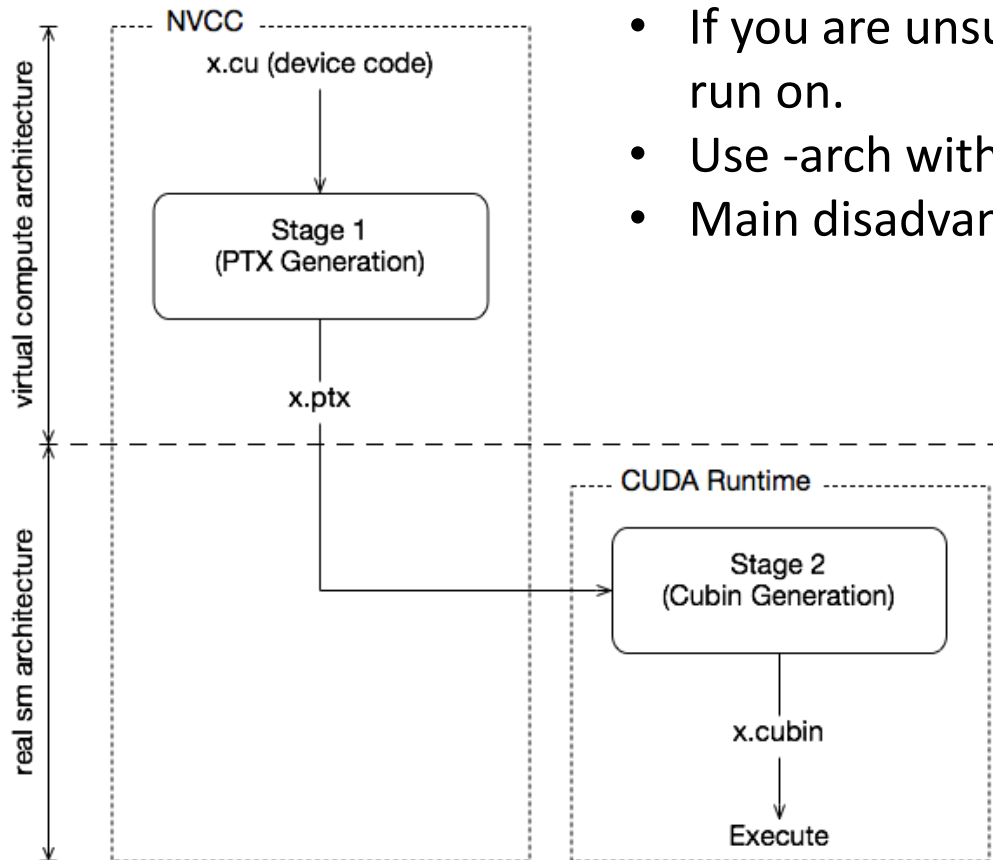
- If you are unsure which exact GPU the code will run on.
- Use -arch without -code
- Main disadvantage: slower startup

Fatbinaries

nvcc x.cu -arch=compute_30 -code=compute_30,sm_30,sm_35

generate PTX
and keep it in the
binary generated,
for JIT on future GPUs

Generate binaries for
two versions of Kepler

At runtime, the CUDA driver will select the most appropriate translation
when the device function is launched.

Till now we have single virtual architecture and several real architectures.
How about several virtual architectures?

--generate-code

```
nvcc x.cu \
--generate-code arch=compute_20,code=sm_20 \
--generate-code arch=compute_20,code=sm_21 \
--generate-code arch=compute_30,code=sm_30
```

The Default

nvcc x.cu

is equivalent to

nvcc x.cu -arch=compute_20 -code=sm_20,compute_20

nvcc

- Some nvcc features: --ptxas-options=-v
  - Print the smem, register and other resource usages

- Generates CUDA binary file: nvcc –cubin
  - cubin file is the cuda executable

Dealing with binary files

	cuobjdump	nvdisasm

Extract ptx and extract and disassemble cubin from the following input files:
 	
▸ Host binaries
 	
    ▸ Executables
    ▸ Object files
    ▸ Static libraries
▸ External fatbinary files
	Yes	No

Control flow analysis and output	No	Yes
Advanced display options	No	Yes

nvprof

- CUDA profiler: profiling data from the command line

```
$ nvprof [nvprof_args] <app> [app_args]
```

- To profile a region of the application:
  1. #include <cuda_profiler_api.h>
  2. in the host function surround the region with:
     - cudaProfilerStart()
     - cudaProfilerStop()
  3. nvcc myprog.cu
  4. nvprof --profile-from-start-off ./a.out

nvprof summary mode (default)

```
$ nvprof dct8x8

======== Profiling result:
 Time(%)      Time  Calls       Avg       Min       Max  Name
   49.52    9.36ms    101   92.68us   92.31us   94.31us  CUDAkernel2DCT(float*, float*, int)
   37.47    7.08ms     10  708.31us  707.99us  708.50us  CUDAkernel1DCT(float*,int, int,int)
    3.75  708.42us      1  708.42us  708.42us  708.42us  CUDAkernel1IDCT(float*,int,int,int)
    1.84  347.99us      2  173.99us  173.59us  174.40us  CUDAkernelQuantizationFloat()
    1.75  331.37us      2  165.69us  165.67us  165.70us  [CUDA memcpy DtoH]
    1.41  266.70us      2  133.35us   89.70us  177.00us  [CUDA memcpy HtoD]
    1.00  189.64us      1  189.64us  189.64us  189.64us  CUDAkernelShortDCT(short*, int)
    0.94  176.87us      1  176.87us  176.87us  176.87us  [CUDA memcpy HtoA]
    0.92  174.16us      1  174.16us  174.16us  174.16us  CUDAkernelShortIDCT(short*, int)
    0.76  143.31us      1  143.31us  143.31us  143.31us  CUDAkernelQuantizationShort(short*)
    0.52   97.75us      1   97.75us   97.75us   97.75us  CUDAkernel2IDCT(float*, float*)
    0.12   22.59us      1   22.59us   22.59us   22.59us  [CUDA memcpy DtoA]
```

nvprof trace mode

```
$ nvprof --print-gpu-trace dct8x8

======== Profiling result:
    Start   Duration  Grid Size   Block Size  Regs  SSMem  DSMem  Size    Throughput  Name
  167.82ms  176.84us  -           -           -     -      -      1.05MB  5.93GB/s    [CUDA memcpy HtoA]
  168.00ms  708.51us  (64 64 1)   (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  168.95ms  708.51us  (64 64 1)   (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  169.74ms  708.26us  (64 64 1)   (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  170.53ms  707.89us  (64 64 1)   (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  171.32ms  708.12us  (64 64 1)   (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  172.11ms  708.05us  (64 64 1)   (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  172.89ms  708.38us  (64 64 1)   (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  173.68ms  708.31us  (64 64 1)   (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  174.47ms  708.15us  (64 64 1)   (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  175.26ms  707.95us  (64 64 1)   (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  176.05ms  173.87us  (64 64 1)   (8 8 1)     27    0B     0B     -       -           CUDAkernelQuantization (…)
  176.23ms   22.82us  -           -           -     -      -      1.05MB  45.96GB/s   [CUDA memcpy DtoA]
```

GPU-trace mode provides a timeline of all activities taking place on the GPU in chronological order.

Print individual kernel invocations and sort them in chronological order

Print CUDA runtime/driver API trace

```
$ nvprof --print-gpu-trace --print-api-trace dct8x8

======== Profiling result:
     Start   Duration  Grid Size  Block Size  Regs  SSMem  DSMem  Size    Throughput  Name
  167.82ms   176.84us  -          -           -     -      -      1.05MB  5.93GB/s    [CUDA memcpy HtoA]
  167.81ms     2.00us  -          -           -     -      -      -       -           cudaSetupArgument
  167.81ms    38.00us  -          -           -     -      -      -       -           cudaLaunch
  167.85ms     1.00ms  -          -           -     -      -      -       -           cudaDeviceSynchronize
  168.00ms   708.51us  (64 64 1)  (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
  168.86ms     2.00us  -          -           -     -      -      -       -           cudaConfigureCall
  168.86ms     1.00us  -          -           -     -      -      -       -           cudaSetupArgument
  168.86ms     1.00us  -          -           -     -      -      -       -           cudaSetupArgument
  168.86ms     1.00us  -          -           -     -      -      -       -           cudaSetupArgument
  168.87ms       0ns  -          -           -     -      -      -       -           cudaSetupArgument
  168.87ms    24.00us  -          -           -     -      -      -       -           cudaLaunch
  168.89ms   761.00us  -          -           -     -      -      -       -           cudaDeviceSynchronize
  168.95ms   708.51us  (64 64 1)  (8 8 1)     28    512B   0B     -       -           CUDAkernel1DCT(float*, …)
```

nvcc --devices *x* --events *y* ./a.out

- x: device number in case of multi-GPU
- y: event name
  - Gives very useful information, such as:
    - number of global memory loads, stores, …
    - number of global memory coalesced
    - branch divergences

```
$ nvprof --devices 0 --events branch,divergent_branch dct8x8
```

Revisiting Shared Memory

Till now

- We know that we have to use __shared__
- But, you leave it to the runtime system to distribute shared memory among blocks. Can you change that?
- What if you don't know how much shared memory we need before execution?

This is what we know

```
__global__ void staticShared(int
*d, int n)
{
    __shared__ int array[64];
    int t = threadIdx.x;
    s[t] = d[t];
}
```

What if you don't know this size beforehand?

Dynamic Shared Memory

```
__global__ void DynamicShared(int
*d, int n)
{
    extern __shared__ int array[];
    int t = threadIdx.x;
    s[t] = d[t];
}
main()
{
  ...
  DynamicShared<<<100, 100, N*sizeof(int)>>>(int *, int);

}
```

Step 1:
A different declaration with no size in the kernel

Step 2:
The 3rd argument of kernel launch is the shared memory allocation size per thread block in bytes.

What if you need more than one array in the shared memory?

- Declare a single extern unsized array (as in the previous slide).

- Use pointers into it to divide it into multiple arrays.

- Example:

```
extern __shared__ int s[];
int *integerData = s;                        // nI ints
float *floatData = (float*)&integerData[nI]; // nF floats
char *charData = (char*)&floatData[nF];      // nC chars
```

Potential Performance Loss: Shared Memory Bank Conflicts

- Shared memory is divided into equally sized memory modules called banks.

- Banks can be accessed simultaneously.

- Shared memory accesses that span b distinct banks yield an effective bandwidth that is b times as high as the bandwidth of when accesses map to the same bank.

- Exceptions are: broadcast and multicast.

Potential Performance Loss: Shared Memory Bank Conflicts

- Shared memory banks are organized such that successive 32-bit words are assigned to successive banks.

- Bandwidth is 32 bits per bank per clock cycle.

- Warp size is 32

Potential Performance Loss: Shared Memory Bank Conflicts

- Now, depending on compute capability:
  - 1.x: the number of banks is 16. A shared memory request for a warp is split into one request for the first half of the warp and one request for the second half of the warp.
  - 2.x: the number of banks is 32. A shared memory request for a warp is not split.
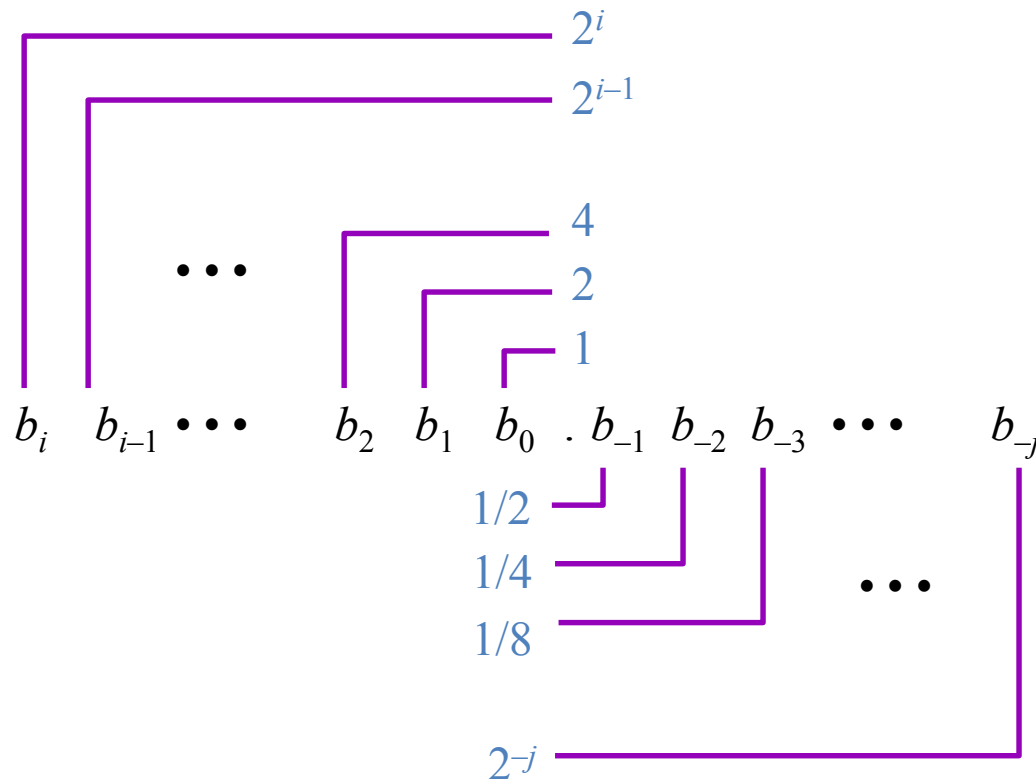 - 3.x: have configurable bank size

cudaDeviceSetSharedMemConfig(*i*)

- *i above can be:*
  - cudaSharedMemBankSizeFourByte ← default
  - cudaSharedMemBankSizeEightByte

Floating Points

Importance of Floating Points

- Many graphics operations are floating point operations
- GPU performance is measure in GFLOPS

2^i

2^{i-1}

4

2

1

$b_i \quad b_{i-1} \; \cdots \quad b_2 \quad b_1 \quad b_0 \; . \; b_{-1} \; b_{-2} \; b_{-3} \; \cdots \quad b_{-j}$

$1/2$

$1/4$

$1/8$

2^{-j}

Turing Award 1989 to William Kahan for design of the IEEE Floating Point Standards 754 (binary) and 854 (decimal)

What is Excel doing?

A1: | 1.333333333333330000 | =4/3 |

Floating Point

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576×10^9

- Representation:
  - sign, exponent, mantissa: $(-1)^{sign} \times$ mantissa $\times 2^{exponent}$
  - more bits for mantissa gives more accuracy
  - more bits for exponent increases range

- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit mantissa
  - double precision: 11 bit exponent, 52 bit mantissa

IEEE 754 floating-point standard

- Leading "1" bit of significand is implicit
  (called **hidden 1 technique,** except when exp = -127**)**
- Exponent is "biased" to make sorting easier
  - all 0s is smallest exponent
  - all 1s is largest exponent
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{sign} \times (1+significand) \times 2^{exponent - bias}$

- Example:

  - decimal: -.75 = - ($\frac{1}{2}$ + $\frac{1}{4}$)
  - binary: -.11 = -1.1 x 2^{-1}
  - floating point: exponent = 126 = 01111110

  - IEEE single precision: 10111111010000000000000000000000

More about IEEE floating Point Standard

Single Precision:

$$(-1)^{\text{sign}} \times (1 + \text{mantissa}) \times 2^{\text{exponent} - 127}$$

The variables shown in red are the numbers stored in the machine

Floating Point Example

what is the decimal equivalent of

1   01110110   10110000...0

Based on exp
we have 3 encoding schemes

- exp ≠ 0..0 or 11…1 → normalized encoding
- exp = 0… 000 → denormalized encoding
- exp = 1111…1 → special value encoding
  - frac = 000…0
  - frac = something else

1. Normalized Encoding

- Condition: exp ≠ 000...0 and exp ≠ 111...1

referred to as Bias

- Exponent is: $E = Exp - (2^{k-1} - 1)$, k is the # of exponent bits
  - Single precision: $E = exp - 127$
  - Double precision: $E = exp - 1023$

frac

Range(E)=[-126,127]
Range(E)=[-1022,1023]

- Significand is: $M = 1.xxx\ldots x_2$
  - Range(M) = [1.0, 2.0-ε)
  - Get extra leading bit for free

Normalized Encoding Example

- Value: `Float F = 15213.0;`
  - 15213_{10} = 11101101101101_2
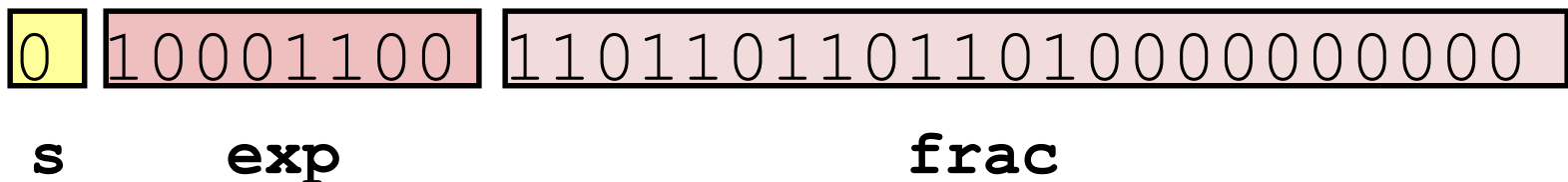   - = $1.1101101101101_2 \times 2^{13}$

- Significand
  - M = $1.\underline{1101101101101}_2$
  - **frac**= $\underline{1101101101101}0000000000_2$

- Exponent
  - E = exp – Bias = exp - 127 = 13
  - ➔ exp = 140 = 10001100_2

- Result:

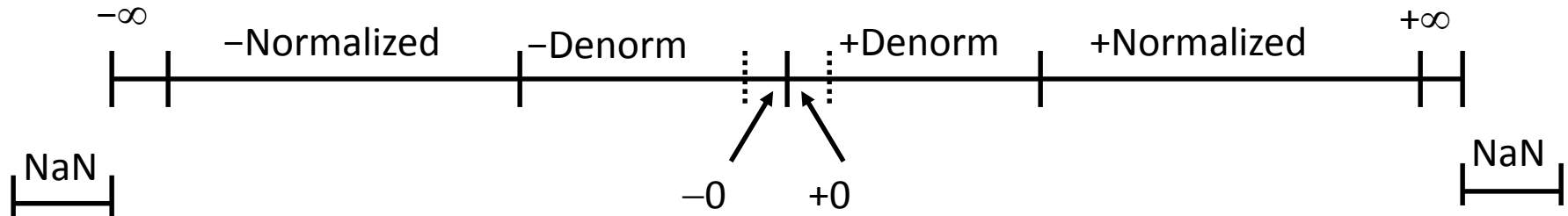| 0	10001100	11011011011010000000000
s	**exp**	**frac**

2. Denormalized Encoding

- Condition: exp = 000…0

- Exponent value: $E = 1 - Bias$ (instead of $E = 0 - Bias$)
- Significand is: $M = 0.xxx…x_2$ (instead of $M=1.xxx_2$)

  $\underbrace{xxx…x}_{\text{frac}}$

- Cases
  - **exp** = 000…0, **frac** = 000…0
    - Represents zero
    - Note distinct values: +0 and –0
  - **exp** = 000…0, **frac** ≠ 000…0
    - Numbers very close to 0.0

3. Special Values Encoding

- Condition: **exp** = 111…1

- Case: **exp** = 111…1, **frac** = 000…0
  - Represents value ∞ (infinity)
  - Operation that overflows
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- Case: **exp** = 111…1, **frac** ≠ 000…0
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(−1), ∞ − ∞, $\infty \times 0$

Visualization: Floating Point Encodings

Floating Point: IEEE 754

What is the decimal equivalent of:

10011111110101000000000000000000000

-

127
So:

•**Real exponent = 127 -127 = 0**
•**There is hidden 1**

1.1010....0
= 1.625

Final answer = -1.625

In Summary About Floating Points

exponent	mantissa	meaning
11...1	≠ 0	NaN
11...1	=0	$(-1)^S * \infty$
00...0	≠0	denormalized
00...0	=0	0

Algorithm Considerations

- Non representable numbers are rounded
- This rounding *error* leads to different results depending on the order of operations
  - Non-repeatability makes debugging harder
- A common technique to maximize floating point arithmetic accuracy is to presort data before a reduction computation.

So..

When doing floating-point operations in parallel you have to decide:

- How much accuracy is good enough?

- Do you need single-precision or double precision?

- Can you tolerate presorting overhead, if you care about rounding errors?

Streams

Streams

- A sequence of operations that execute on the device in the order in which they are issued by the host code
- Operations in different streams can be interleaved and, when possible, they can even run concurrently.
- A stream can be sequence of kernel launches and host-device memory copies
- Can have several open streams to the same device at once
- Need GPUs with concurrent transfer/execution capability
- Potential performance improvement: can overlap transfer and computation

Streams

- By default all transfers and kernel launches are assigned to stream 0
  - This means they are executed in order

host thread

FIFO
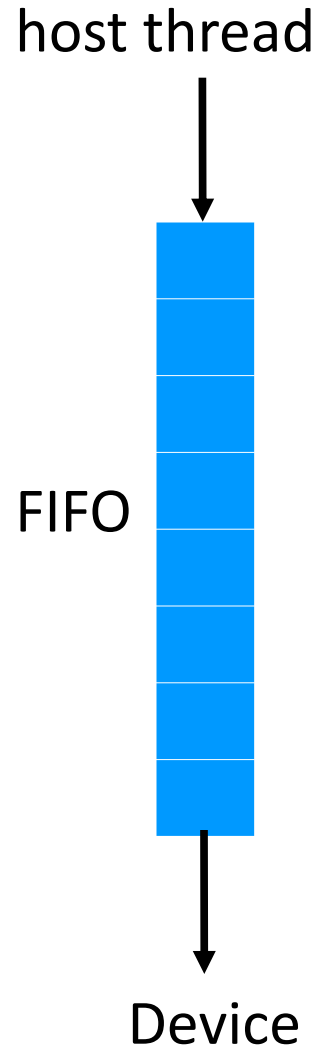
Device

Example: Default Stream

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
    increment<<<1,N>>>(d_a) ;
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- In the code above, from the perspective of the device, all three operations are issued to the same (default) stream and will execute in the order that they were issued.
- From the perspective of the host:
    - data transfers are blocking or synchronous transfers
    - kernel launch is asynchronous.

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a) ;
anyCPUfunction();
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Example: Non-Default Stream

Non-default streams in CUDA C/C++ are declared, created, and destroyed in host code as follows:

```
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1);
result = cudaStreamDestroy(stream1);
```

To issue data transfer to non-default stream (non-blocking):

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
```

To launch a kernel to non-default stream:

```
increment<<<1,N,0,stream1>>>(d_a);
```

Important

- All operations to non-default streams are non-blocking with respect to the host code.
- Sometimes you need to synchronize the host code with operations in a stream.
- You have several options:
  - cudaDeviceSynchronize() → blocks host
  - cudaStreamSynchronize(stream) → blocks host
  - cudaStreamQuery(stream) → does not block host

Streams

- The amount of overlap execution between two streams depends on:
  - Device supports overlap transfer and kernel execution (compute capability 1.1 and higher)
  - Devices supports concurrent kernel execution (compute capability 2.x and higher)
  - Device supports concurrent data transfer (compute capability 2.x and higher)
  - The order on which commands are issued to each stream

Using streams to overlap device execution with data transfer

- Conditions to be satisfied first:
  - The device must be capable of *concurrent copy and execution.*
  - The kernel execution and the data transfer to be overlapped must both occur in *different, non-default* streams.
  - The host memory involved in the data transfer must be *pinned* memory.

Using streams to overlap device execution with data transfer

```
for (int i = 0; i < nStreams; ++i) {

        int offset = i * streamSize;

        cudaMemcpyAsync(&d_a[offset], &a[offset],
                            streamBytes,
                            cudaMemcpyHostToDevice,
                            stream[i]);

        kernel<<…….>>(d_a, offset);

        cudaMemcpyAsync(&a[offset],  &d_a[offset],
                            streamBytes,
                            cudaMemcpyDeviceToHost,
                            stream[i]);
}
```

So..

- Streams are a good way to overlap execution and transfer, hardware permits.

- Don't confuse kernels, threads, and streams.

Pinned Pages

- Allocate page(s) from system RAM (cudaMallocHost() or cudaHostAlloc())
  - Accessible by device (but wait till next slide)
  - Cannot be paged out
  - Enables highest memory copy performance (cudaMemcpyAsync())
  - Don't forget cudaFreeHost();
- If too much pinned pages, overall system performance may greatly suffer.

Host page accessible by the device??

- The pointer to the host memory is not directly transferable to device, except with:

  – cudaHostGetDevicePointer(void ** pDevice,
                                        void * pHost,
                                unsigned int flags)

  – flags are 0 for now

- Accessing host memory from device without explicit copy is called "zero-copy" mechanism.

Steps for Zero-Copy

1. cudaHostAlloc (void ** ptr, size_t size, unsigned int flags)
   – flag here: cudaHostAllocMapped
2. cudaHostGetDevicePointer()
3. Then use the pointer in your kernel on device as if it is in the GPU memory

```c
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#define N 32        // size of vectors

__global__ void add(int *a,int *b, int *c) {
    int tid = blockIdx.x *  blockDim.x + threadIdx.x;
     if(tid < N)  c[tid] = a[tid]+b[tid];
}

int main(int argc, char *argv[])  {
    int T = 32, B = 1;                      // threads per block and blocks per grid
    int  *a,*b,*c;                          // host pointers
    int *dev_a, *dev_b, *dev_c;             // device pointers to host memory

    cudaHostAlloc( (void**)&a, size, cudaHostAllocMapped);
    cudaHostAlloc( (void**)&b, size, cudaHostAllocMapped);
    cudaHostAlloc( (void**)&c, size, cudaHostAllocMapped );

    …   // load arrays with some numbers

    cudaHostGetDevicePointer(&dev_a, a, 0);
    cudaHostGetDevicePointer(&dev_b, b, 0);
    cudaHostGetDevicePointer(&dev_c ,c, 0);

    add<<<B,T>>>(dev_a,dev_b,dev_c);

    cudaFreeHost(a);
    cudaFreeHost(b);
    cudaFreeHost(c);

    return 0;
}
```

So..

- If the CPU program requires a lot of memory, then pinned pages is not a good idea.

Asynchronous Execution

- Asynchronous = returns to host right-away and does not wait for device
- This includes (but not limited to):
  - Kernel launches;
  - Memory copies between two addresses to the same device memory;
  - Memory copies from host to device of a memory block of 64 KB or less;
  - Memory copies performed by functions that are suffixed with Async;

Asynchronous Execution

- Some CUDA API calls and all kernel launches are asynchronous with respect to the host code.

- This means error-reporting is also asynchronous.

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

Other Sources of Concurrency

- Some devices of compute capability 2.x and higher can execute multiple kernels concurrently.
- The maximum number of kernel launches that a device can execute concurrently is 32 on devices of compute capability 3.5 and 16 on devices of lower compute capability.
- A kernel from one CUDA context cannot execute concurrently with a kernel from another CUDA context.
- Kernels that use many textures or a large amount of local memory are less likely to execute concurrently with other kernels.
- Some devices of compute capability 2.x and higher can perform a copy from page-locked host memory to device memory concurrently with a copy from device memory to page locked host memory.

Conclusions

- As we program GPUs we need to pay attention to several performance bottlenecks:
  - Branch diversion
  - Global memory latency
  - Global memory bandwidth
  - Shared memory bank conflicts
  - Communication
  - Limited resources
- We have several techniques in our arsenal to enhance performance
  - Try to make threads in the same warp follow the same control flow
  - Tiling
  - Coalescing
  - Loop unrolling
  - Increase thread granularity
  - Trade one resource for another
  - Memory access pattern
  - Streams
- Pay attention to interaction among techniques