



CSCI-UA.0480-003
Parallel Computing

Lecture 20: CUDA III

Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

<http://www.mzahran.com>



Quick Exercises

If a CUDA device's SM can take up to 1,536 threads and up to 4 blocks, which of the following block configs would result in the most number of threads in the SM?

- 128 threads/blk
- 256 threads/blk
- 512 threads/blk
- 1,024 threads/blk

Quick Exercises

- For a vector addition, assume that the vector length is 2,000, each thread calculates one output element, and the thread block size 512 threads. How many threads will be in the grid?
- Given the above, how many warps do you expect to have divergence due to the boundary check on the vector length?

Quick Exercises

A CUDA programmer says that if they launch a kernel with only 32 threads in each block, they can leave out the `__syncthreads()` instruction wherever barrier synchronization is needed. Do you think this is a good idea? Explain.

A Motivational Example

- G80 supports 86.4 GB/s of global memory access
- Single precision floating point = 4 bytes
- Then we cannot load more than $86.4/4 = 21.6$ giga single precision data per second
- Theoretical peak performance of G80 is 367gigaflops!

How come??

Computation vs Memory Access

- Compute to global memory access (CGMA) ratio

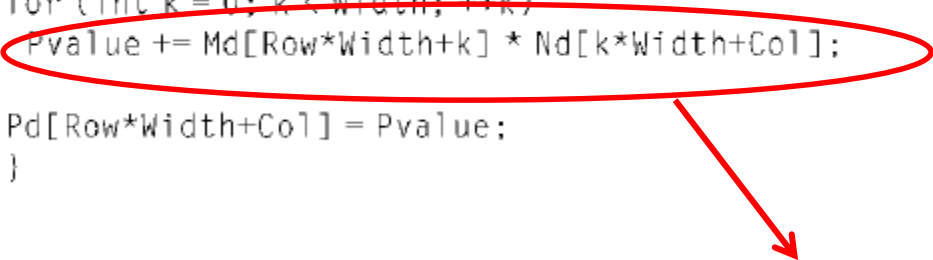
Definition

The number of FP calculations performed for each access to the global memory within a region in a CUDA program.

Computation vs Memory Access

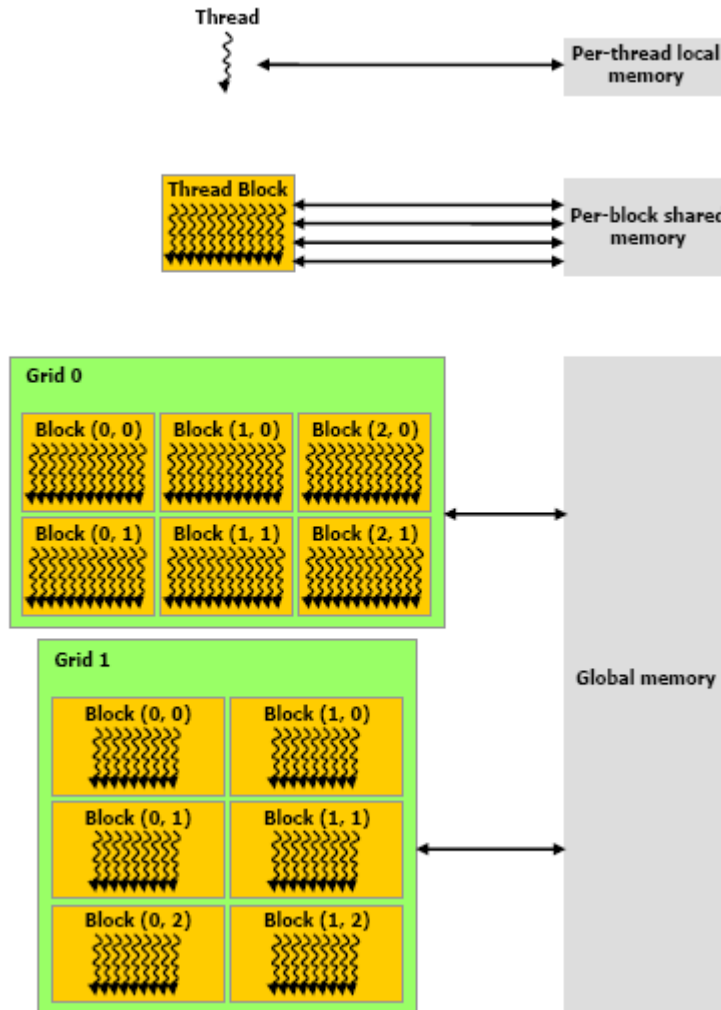
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];
    Pd[Row*Width+Col] = Pvalue;
}
```



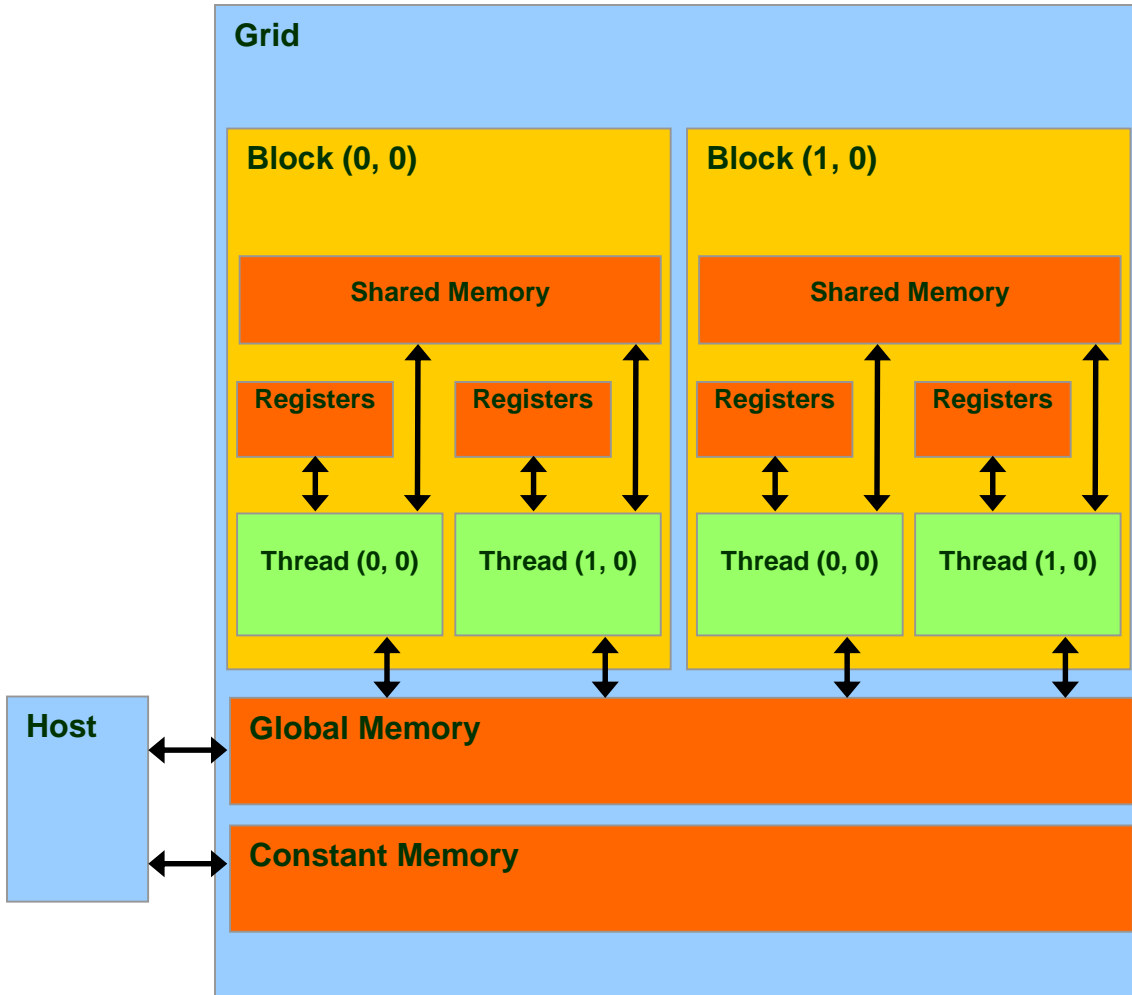
2 memory accesses
1 FP multiplication
1 FP addition
so CGMA = 1

Main Goals for This Lecture



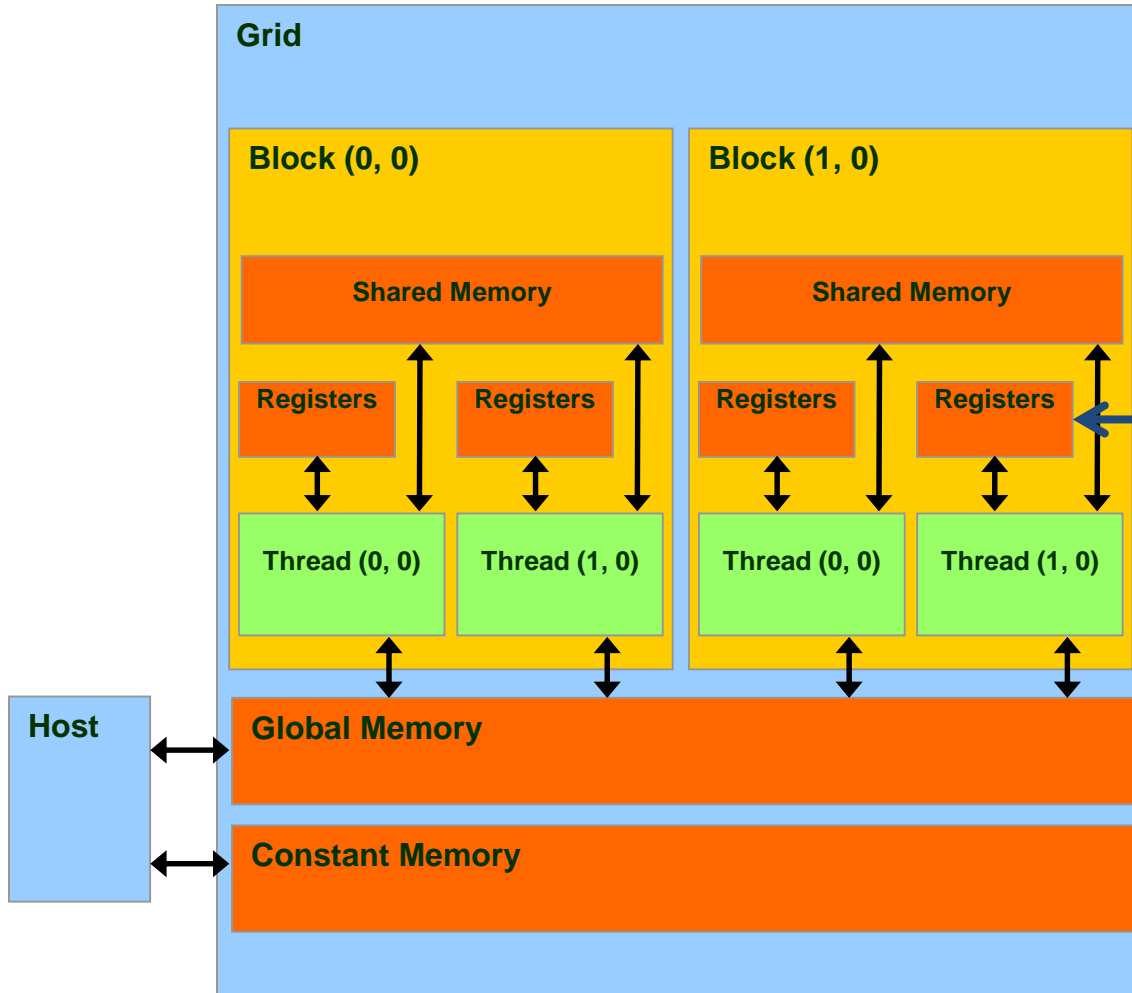
- How to make the best use of the GPU memory system?
- How to deal with hardware limitation?

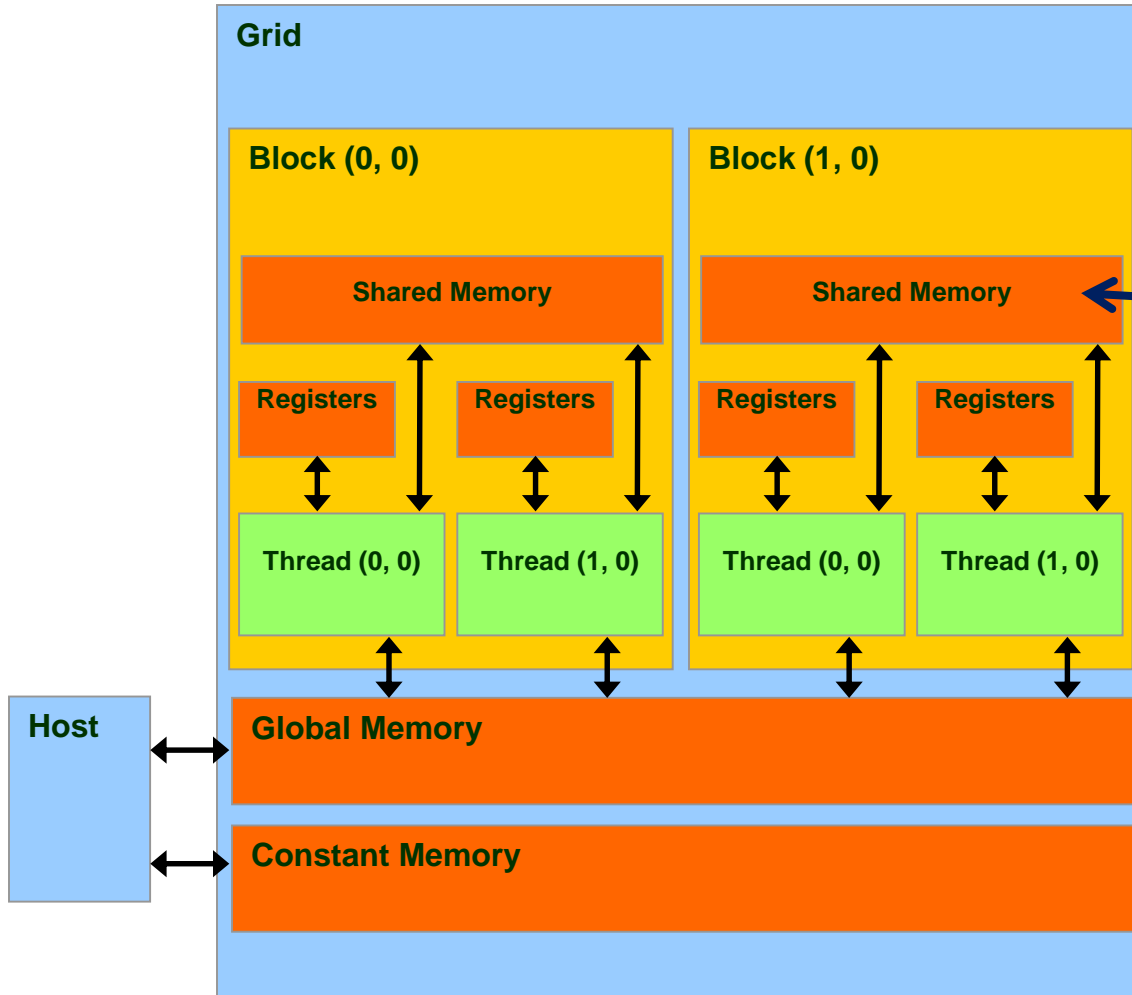
Measure of success: higher CGMA



Registers

- Fastest.
- Do not consume off-chip bandwidth.
- Only accessible by a thread.
- Lifetime of a thread



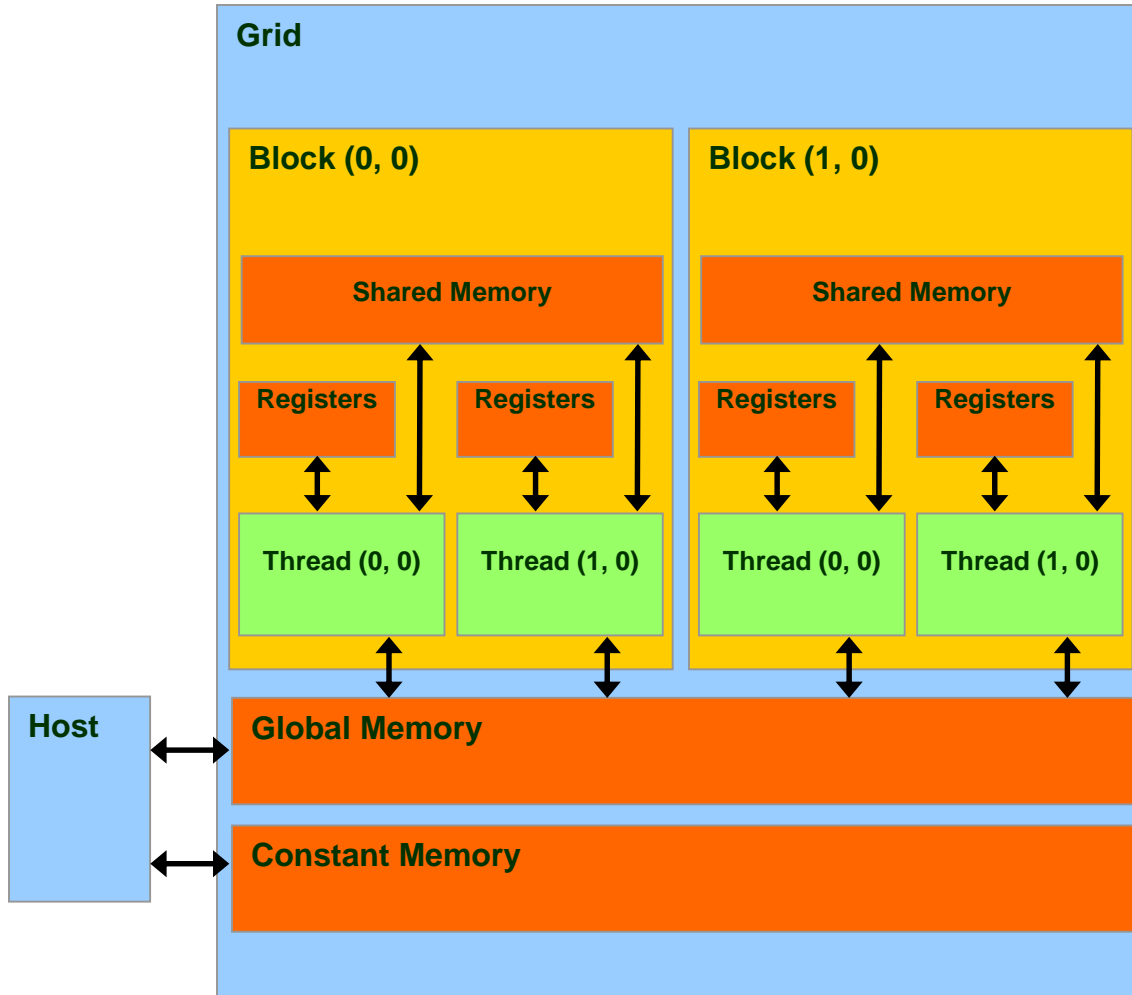


Shared Memory

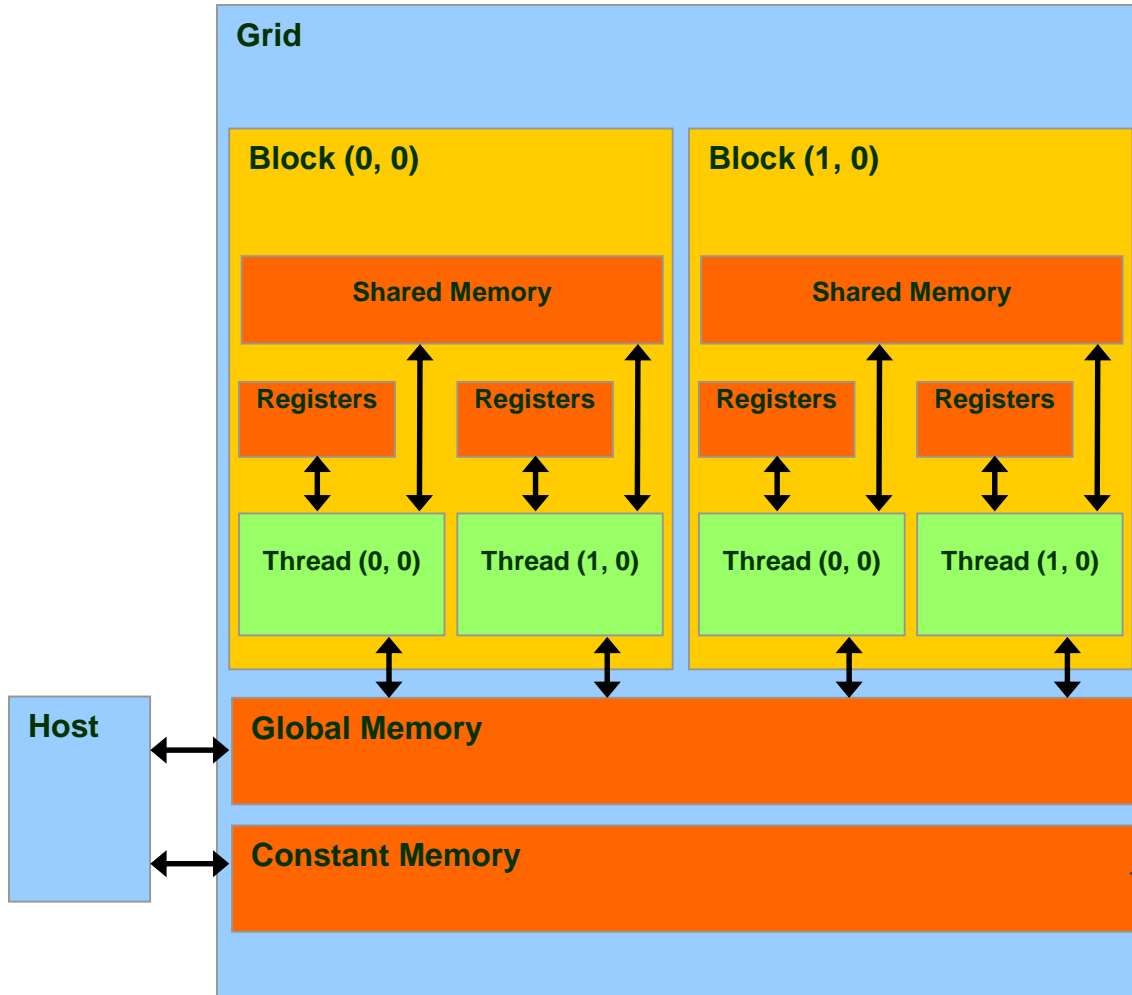
- Extremely fast
- Highly parallel
- Restricted to a block
- Example: Fermi's shared/L1 is 1+TB/s aggregate

Global Memory

- Typically implemented in DRAM
- High access latency: 400-800 cycles
- Finite access bandwidth
- Potential of traffic congestion
- Throughput up to 177GB/s



Traffic congestion prevents all but a few threads from making progress.



Constant Memory

- Read only
- Short latency and high bandwidth when all threads access the same Location
- Small in size ~64KB

Important!

- Each access to registers involves fewer instructions than global memory.
- Aggregate register files bandwidth = ~two orders of magnitude that of the global memory!
- Energy consumed for accessing a value from the register file = ~ at least an order of magnitude lower than accessing global memory!
- Shared memory is part of the address space → accessing it requires load/store instructions.

Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

Scope: the range of threads that can access a variable

Lifetime: the portion of the program's execution when the variable is available for use.

`__device__` is optional when used with `__shared__`, or `__constant__`

Automatic variables reside in a register

Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

Automatic array variables local to a thread reside in **local memory**.

local memory



Does not physically exist. It is an abstraction to the local scope of a thread. Actually put in global memory by the compiler.

Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

The variable must be declared within the kernel function body; and will be available only within the kernel code.

Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

The variable must be declared outside of any function.

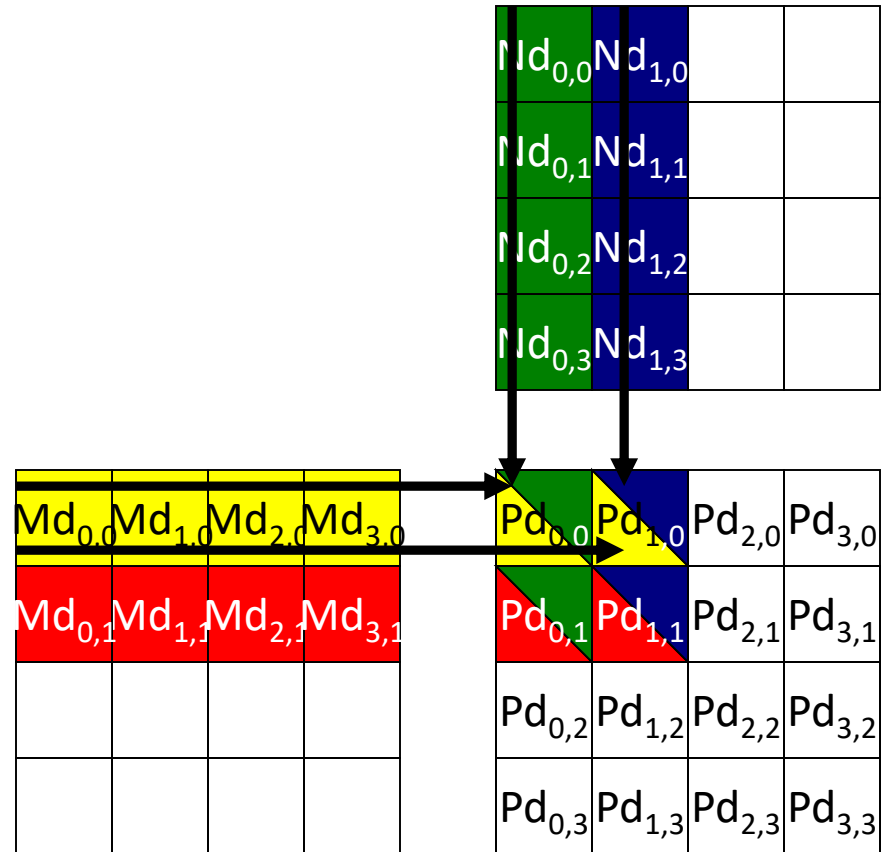
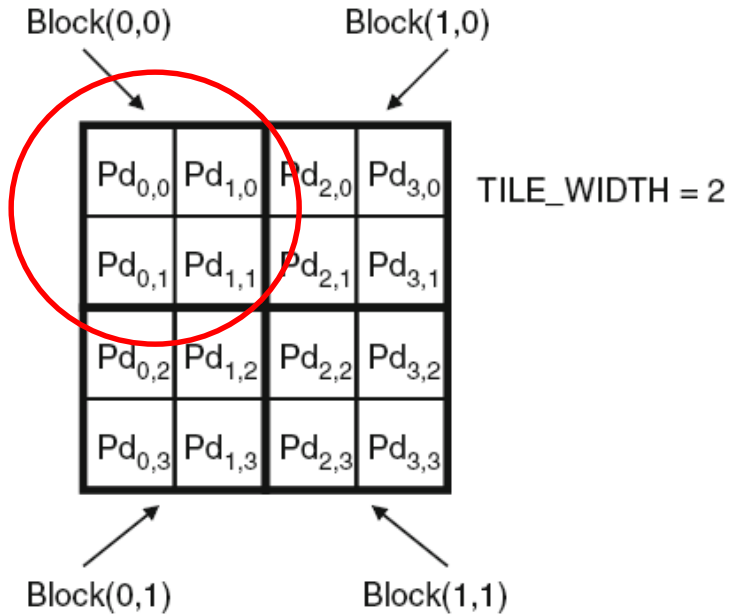
- Declaration of constant variables must be outside any function body.
- Currently total size of constant variables in an application is limited to 64KB.

By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the **visibility** and **access speed** of the variable.

Reducing Global Memory Traffic

- Global memory access is performance bottleneck.
- The lower *CGMA* the lower the performance
- Reducing global memory access enhances performance.
- A common strategy is **tiling**: partition the data into subsets called tiles, such that each tile fits into the shared memory.

Back to Matrix Multiplication



Back to Matrix Multiplication

Access order ↓

$P_{0,0}$ thread _{0,0}	$P_{1,0}$ thread _{1,0}	$P_{0,1}$ thread _{0,1}	$P_{1,1}$ thread _{1,1}
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

Back to Matrix Multiplication

- The basic idea is to make threads that use common elements collaborate.
- Each thread can load different elements into the shared memory before calculations.
- These elements will be used by the thread that loaded them and other threads that share them.

Back to Matrix Multiplication

	Phase 1			Phase 2		
$T_{0,0}$	Md_{0,0} ↓ Mds _{0,0}	Nd_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{1,0} *Nds _{0,1}	Md_{2,0} ↓ Mds _{0,0}	Nd_{0,2} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{1,0} *Nds _{0,1}
$T_{1,0}$	Md_{1,0} ↓ Mds _{1,0}	Nd_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{0,0} *Nds _{1,0} + Mds _{1,0} *Nds _{1,1}	Md_{3,0} ↓ Mds _{1,0}	Nd_{1,2} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{0,0} *Nds _{1,0} + Mds _{1,0} *Nds _{1,1}
$T_{0,1}$	Md_{0,1} ↓ Mds _{0,1}	Nd_{0,1} ↓ Nds _{0,1}	PdValue _{0,1} += Mds _{0,1} *Nds _{0,0} + Mds _{1,1} *Nds _{0,1}	Md_{2,1} ↓ Mds _{0,1}	Nd_{0,3} ↓ Nds _{0,1}	PdValue _{0,1} += Mds _{0,1} *Nds _{0,0} + Mds _{1,1} *Nds _{0,1}
$T_{1,1}$	Md_{1,1} ↓ Mds _{1,1}	Nd_{1,1} ↓ Nds _{1,1}	PdValue _{1,1} += Mds _{0,1} *Nds _{1,0} + Mds _{1,1} *Nds _{1,1}	Md_{3,1} ↓ Mds _{1,1}	Nd_{1,3} ↓ Nds _{1,1}	PdValue _{1,1} += Mds _{0,1} *Nds _{1,0} + Mds _{1,1} *Nds _{1,1}

Time

Back to Matrix Multiplication

- Potential reduction in global memory traffic in matrix multiplication example is proportional to the dimension of the blocks used.
 - With $N \times N$ blocks the potential reduction would be N
- If an input matrix is of dimension M and the tile size is $TILE_WIDTH$, the dot product will be performed in $M/TILE_WIDTH$ phases.

Back to Matrix Multiplication

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x; int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

7.  float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collaborative loading of Md and Nd tiles into shared memory
9.    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.   Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
11.   __syncthreads();

12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.     Pvalue += Mds[ty][k] * Nds[k][tx];
14.   __syncthreads();
    }
15. Pd[Row*Width + Col] = Pvalue;
}
```



The Phases

Back to Matrix Multiplication

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x; int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

7.  float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collaborative loading of Md and Nd tiles into shared memory
9.    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.   Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
11.   __syncthreads();

12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.     Pvalue += Mds[ty][k] * Nds[k][tx];
14.   __syncthreads();
}
15. Pd[Row*Width + Col] = Pvalue;
}
```

→ to be sure needed elements
are loaded

→ to be sure calculations are
completed

Exercise

How can we use shared memory to reduce global memory bandwidth for matrix addition?

Do you Remember the G80 example?

- 86.4 GB/s global memory bandwidth
- In matrix multiplication if we use 16x16 tiles -> reduction in memory traffic by a factor of 16
- Global memory can now support $[(86.4/4) \times 16] = 345.6$ gigaflops -> very close to the peak (367gigaflops).

Memory As Limiting Factor to Parallelism

- Limited shared memory limits the number of threads that can execute simultaneously in SM for a given application
 - The more memory locations each thread requires, the fewer the number of threads per SM
 - Same applies to registers

Memory As Limiting Factor to Parallelism

- **Example: Registers**

- G80 has 8K registers per SM -> 128K registers for entire processor.
- G80 can accommodate up to 768 threads per SM
- To fill this capacity each thread can use only $8K/768 = 10$ registers.
- If each thread uses 11 registers -> threads per SM are reduced -> **per block granularity**
- e.g. if block contains 256 threads the number of threads will be reduced by 256 -> lowering the number of threads/SM from 768 to 512 (i.e. 1/3 reduction of threads!)

Memory As Limiting Factor to Parallelism

- **Example: Shared memory**
 - G80 has 16KB of shared memory per SM
 - SM accommodates up to 8 blocks
 - To reach this maximum each block must not exceed $16\text{KB}/8 = 2\text{KB}$ of memory.
 - e.g. if each block uses 5KB -> no more than 3 blocks can be assigned to each SM

Error Handling

Error Handling

- In a CUDA program, if we suspect an error has occurred during a kernel launch, then we **must explicitly check** for it after the kernel has executed.
- CUDA runtime will respond to questions ... But won't talk without asked!

```
cudaError_t cudaGetLastError(void);
```

- Called by the host
- returns a value encoding the kind of the last error it has encountered
- check for the error only after we're sure a kernel has finished executing → don't forget kernel calls are async!
 - What will you do?

```
#include <stdio.h>
#include <stdlib.h>

__global__ void foo(int *ptr)
{
    *ptr = 7;
}

int main(void)
{
    foo<<<1,1>>>(0);

    // make the host block until the device is finished with foo
    cudaThreadSynchronize();

    // check for error
    cudaError_t error = cudaGetLastError();
    if(error != cudaSuccess)
    {
        // print the CUDA error message and exit
        printf("CUDA error: %s\n", cudaGetErrorString(error));
        exit(-1);
    }

    return 0;
}
```

```
$ nvcc crash.cu -o crash
$ ./crash
CUDA error: unspecified launch failure
```

Same Technique with Synchronous Calls

```
cudaError_t error = cudaMalloc((void**)&ptr,  
                               10000000000000);  
  
if(error != cudaSuccess)  
{  
    // print the CUDA error message and exit  
    printf("CUDA error: %s\n",  
          cudaGetErrorString(error));  
    exit(-1);  
}
```

The output will be:

CUDA error: out of memory

A note about compilation ...
And some useful tools!

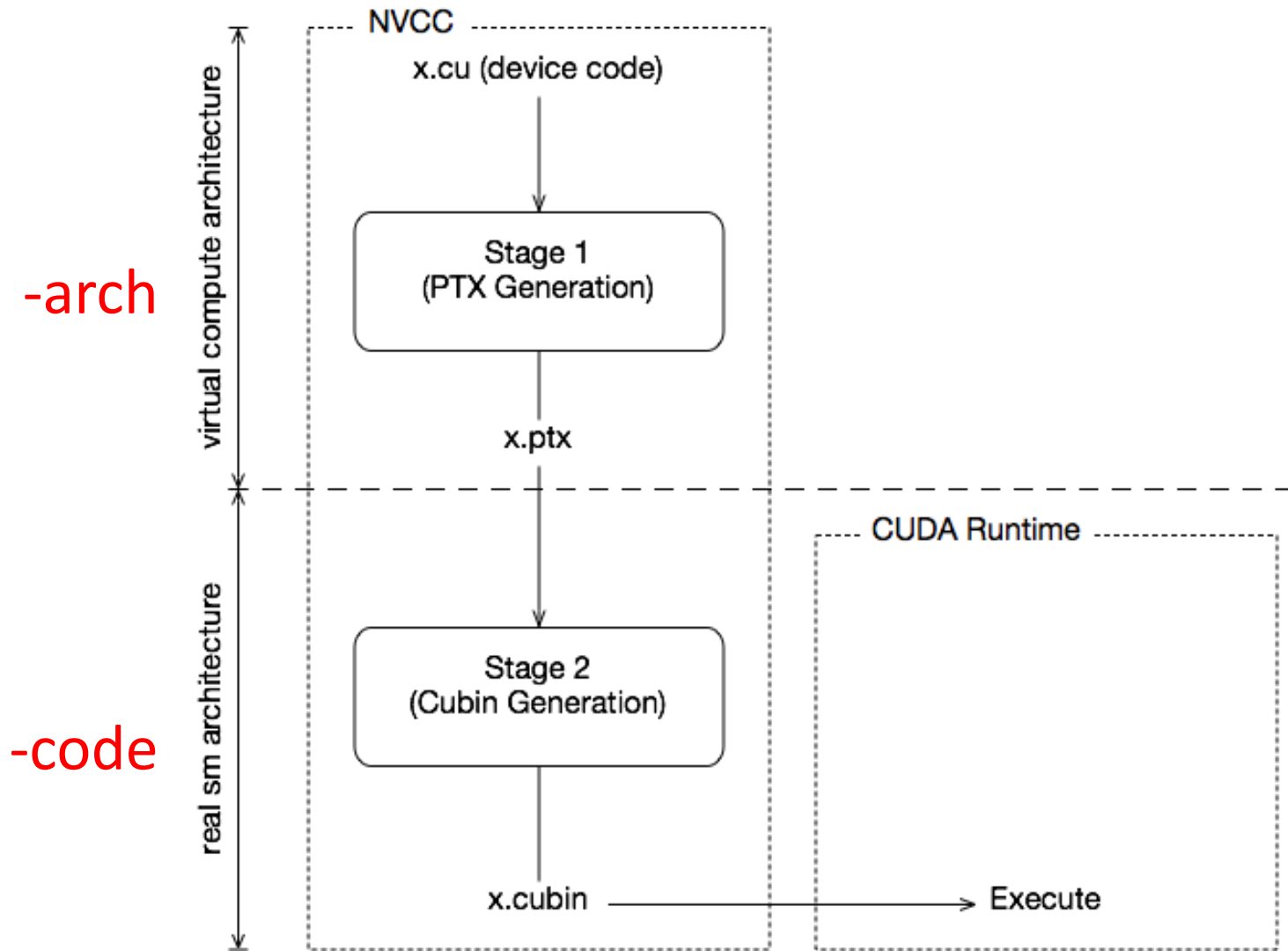
NVCC device specific switches

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

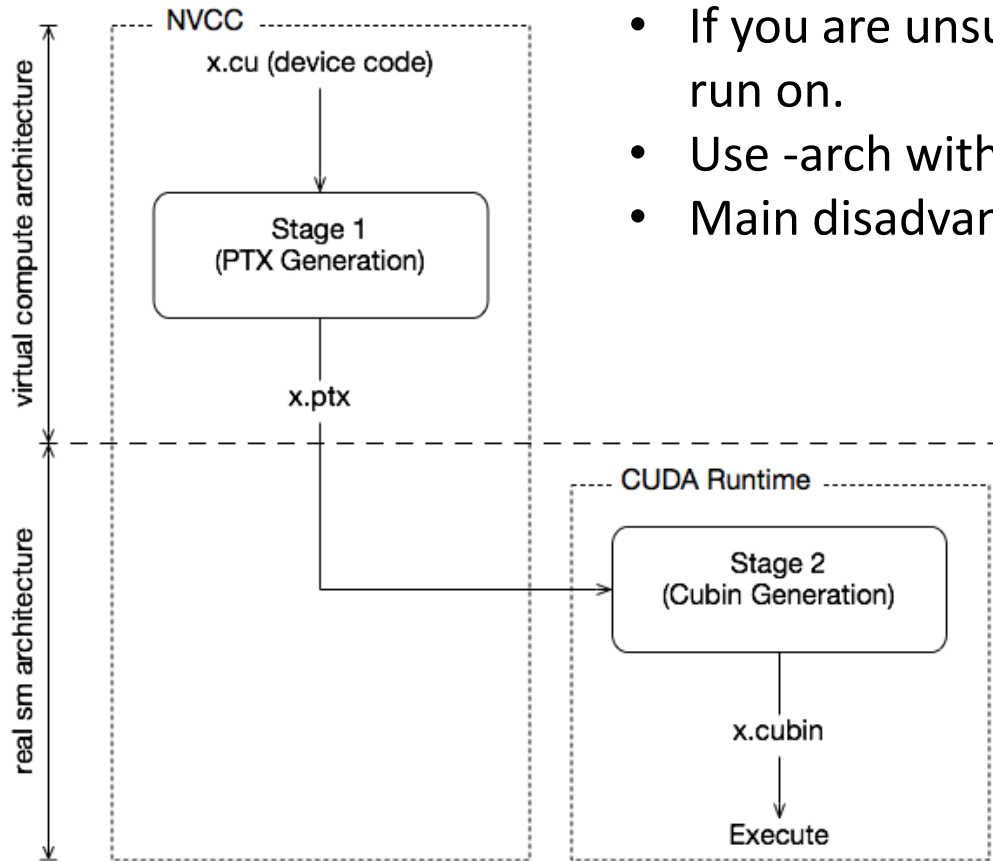
sm_xy

- 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)

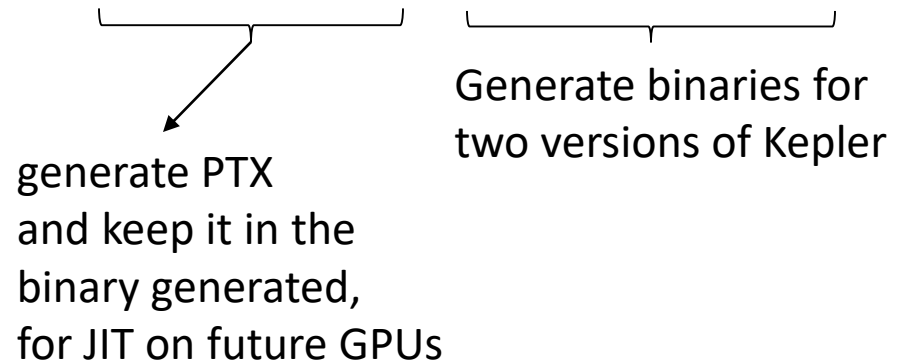
JIT Compilation



- 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
```



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 \
```

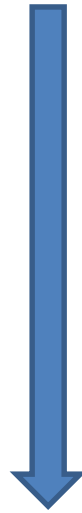
```
--generate-code arch=compute_20,code=sm_20 \
```

```
--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

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]

© NVIDIA Corporation 2012

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*, ...)

```
nvprof --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
```

Conclusions

- Using memory effectively will likely require the redesign of the algorithm.
- The ability to reason about hardware limitations when developing an application is a key concept of **computational thinking**.