



Applied Parallel Computing
parallel-computing.pro

GPU architecture

Aleksei Ivakhnenko

APC



- GPU architecture
 - Kepler
 - Maxwell
- Performance limiters
- Memory
 - Global memory
 - Shared memory
 - Read-only and constant cache
 - Registers and local memory
- Grid configuration
- Arithmetics
 - Branching
 - Intrinsics
- Asynchrony
- Case study



Kepler SMX



Kepler GK110

- 7.1 billion transistors
- 15 SMX
- 6 GB Memory
- > 1 TFLOP FP64
- 1.5 MB L2 cache
- 384-bit GDDR5
- PCI Express 3

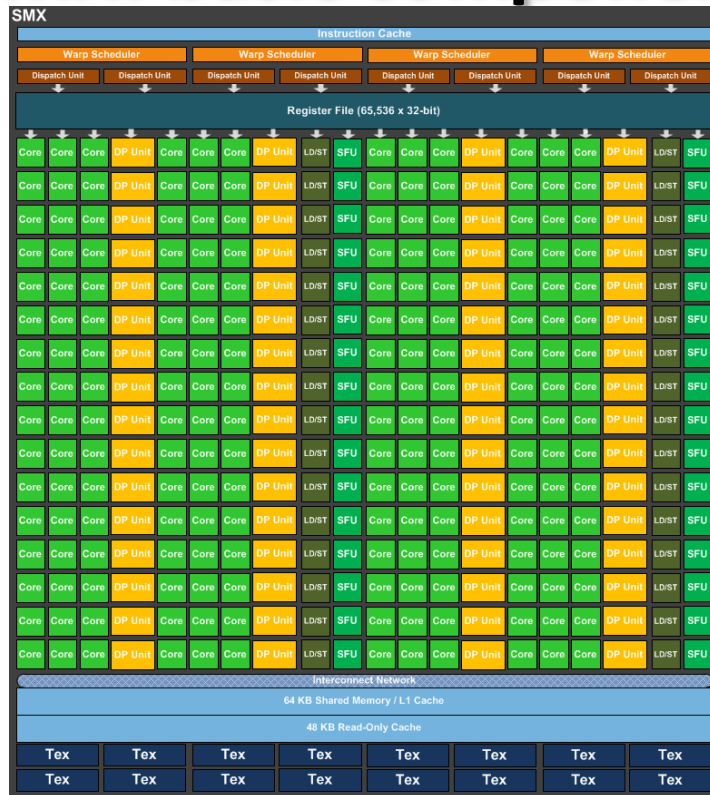




SMX:

- 192 CUDA cores
- 64 DP Unit
- 32 LD/ST Unit
- 32 SFU
- 64 KB L1 cache/ shared memory
- 16 texture processors
- 48 KB read-only cache
- 65536 32-bit register
- 4 warp scheduler
- 8 dispatch units
- Instruction cache

SMX: basic components



Global memory



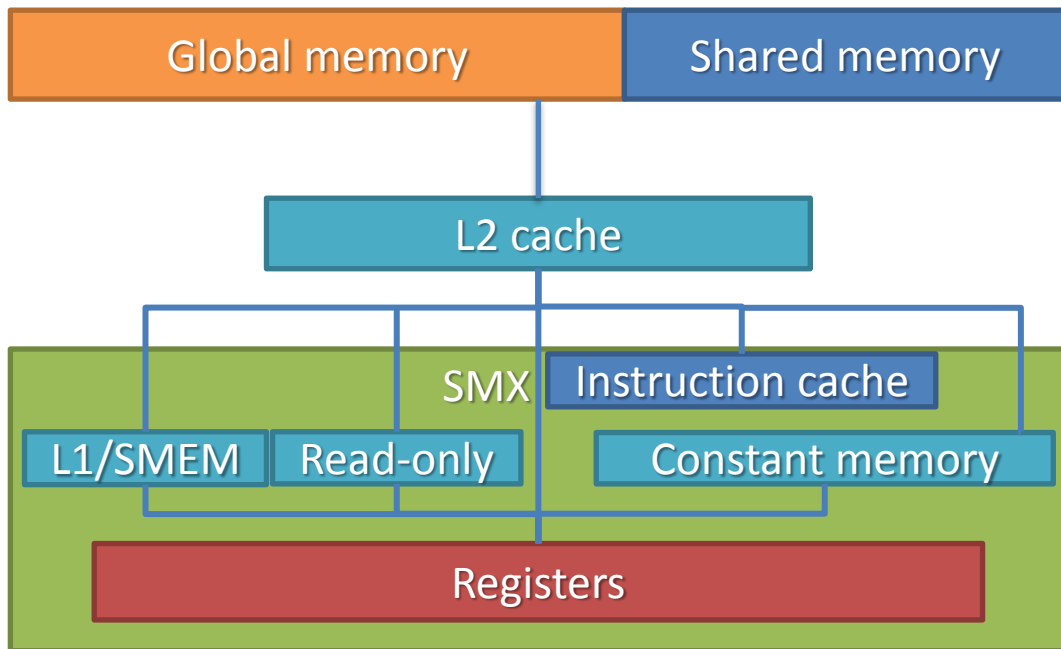
Memory

What can I control:

- Global memory
- Shared memory

Partially:

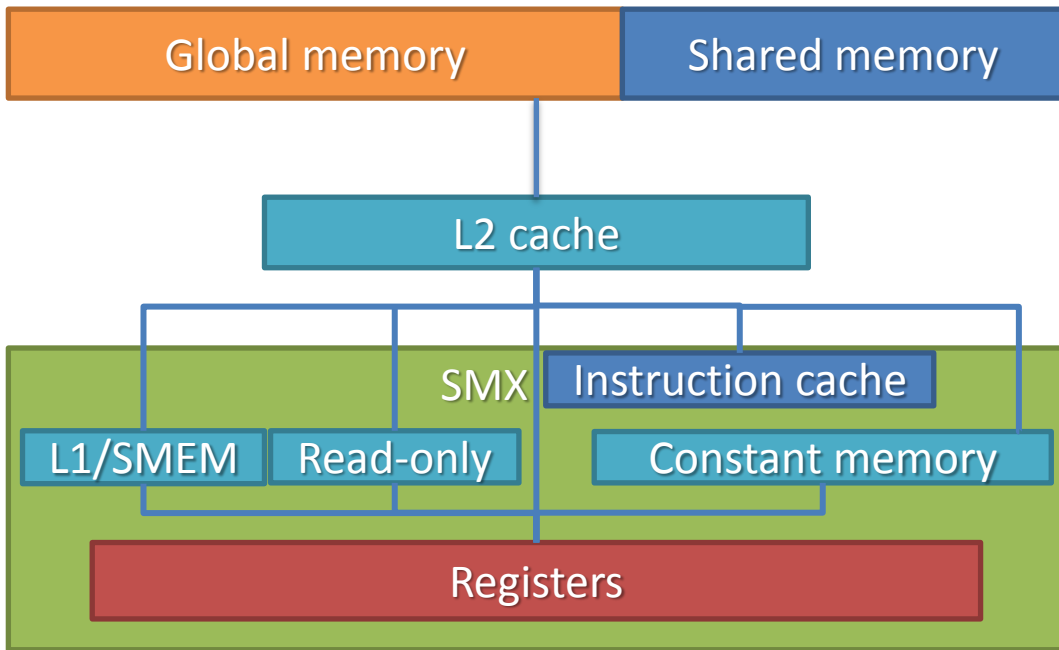
- Local memory
- Registers
- L1 cache
- Read-only cache





Global memory

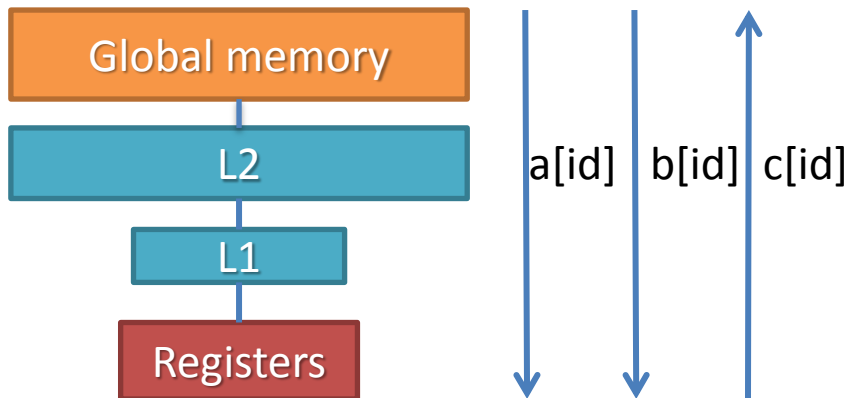
- GDDR5
- Common for the entire device
- Accessible for host using `cudaMemcpy()` and UVA
- Accessible for threads via pointer
- Probable read/write conflicts
- High latency (200-400 cycles)





Global memory requests

```
__global__ void vecAdd(double *a, double *b,  
double *c, int n)  
{  
    int id = blockIdx.x*blockDim.x+threadIdx.x;  
    if (id < n)  
        c[id] = a[id] + b[id];  
}
```

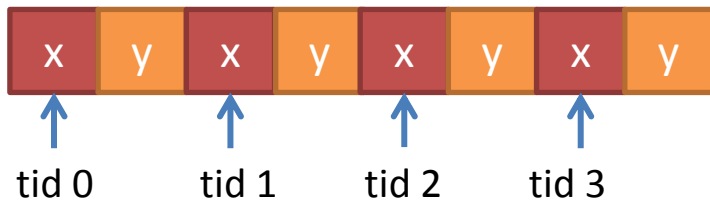


- Requests go through L1 and L2 caches
 - L1 cache line – 128 bytes
 - L2 cache line – 32 bytes
- It is necessary to group requests for high performance
- L1 cache switching on and off:
 - Xptxas -dlcm=ca (on by default)
 - Xptxas -dlcm=cg

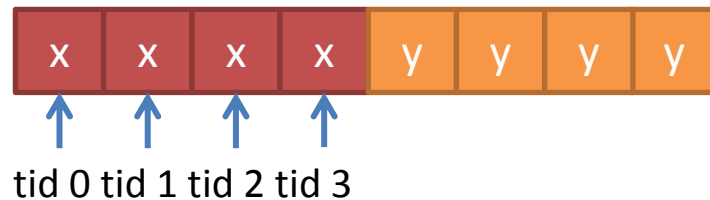


AoS vs SoA

```
struct A{  
    float x;  
    float y;  
};  
struct A myArray[n]
```



```
struct A{  
    float x [n];  
    float y [n];  
};  
struct A myArray
```

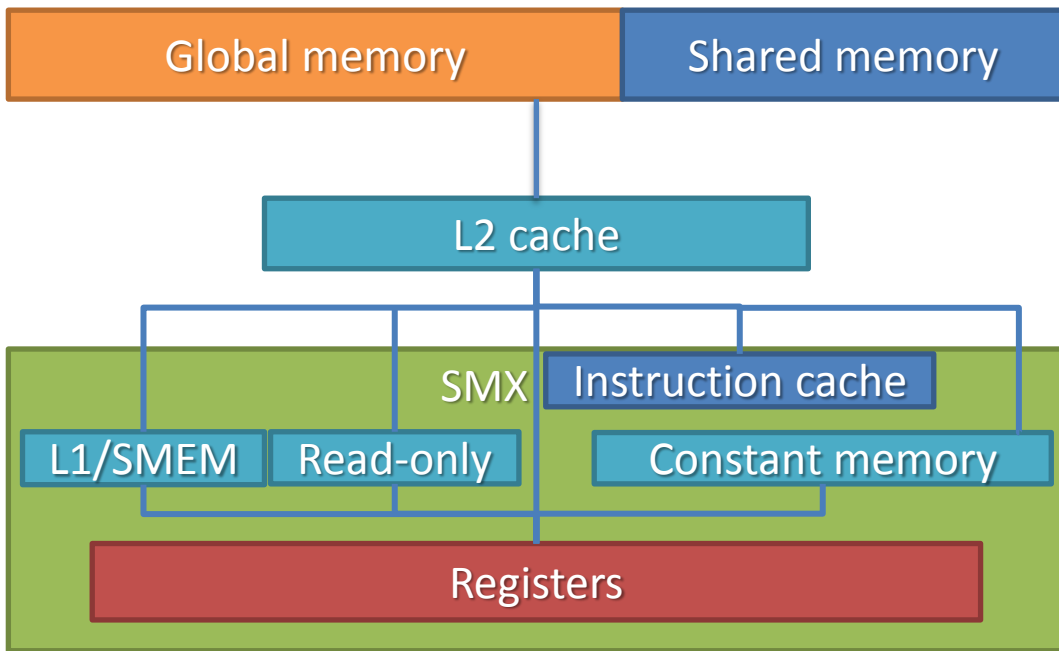


Shared memory



Shared memory

- Allocated per block
- Accessible for all the threads of a block
- Possible read/write conflicts
- Based on the same chip with L1 cache, combinations:
 - 16/48 KB
 - 32/32 KB
 - 48/16 KB
- Latency 20-60 cycles





Shared memory

```
__global__ void staticReverse(int  
*d, int n){  
    __shared__ int s[64];  
    int t = threadIdx.x;  
    int tr = n-t-1;  
    s[t] = d[t];  
    __syncthreads();  
    d[t] = s[tr];  
}
```

- Static or dynamic allocation
- Barrier synchronization using `__syncthreads()`

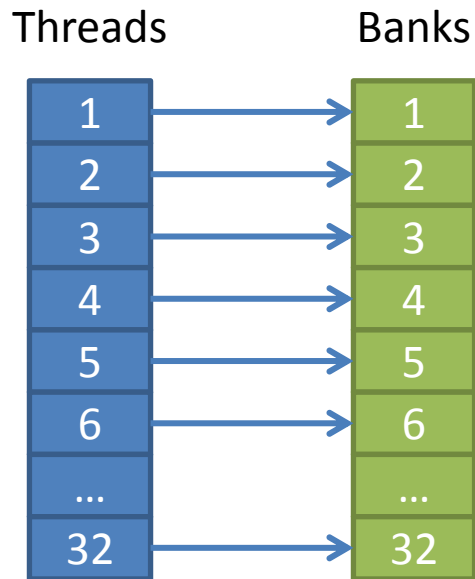


Memory banks

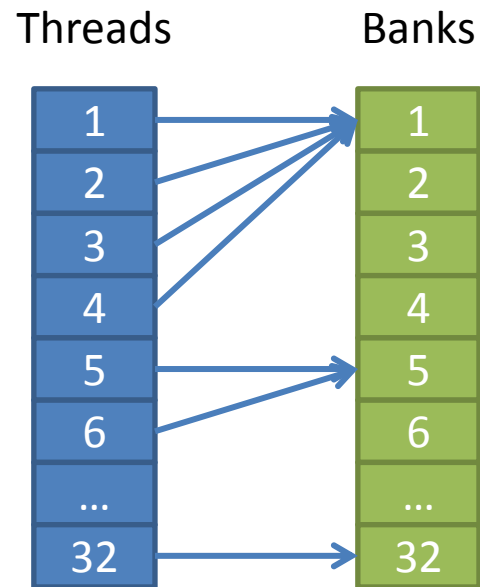
- ④ 32 banks wide of 8 byte
 - Bank conflicts.
 - ✓ Program slows - data requests are serialized
- ④ Two configurations:
 - 4-byte access (by default)
 - 8-byte access
 - Use functions `cudaDeviceSetSharedMemConfig()` and `cudaFuncSetSharedMemConfig()` with the following arguments:
 - ✓ `cudaSharedMemBankSizeFourByte`
 - ✓ `cudaSharedMemBankSizeEightByte`



Bank Conflicts



No conflicts

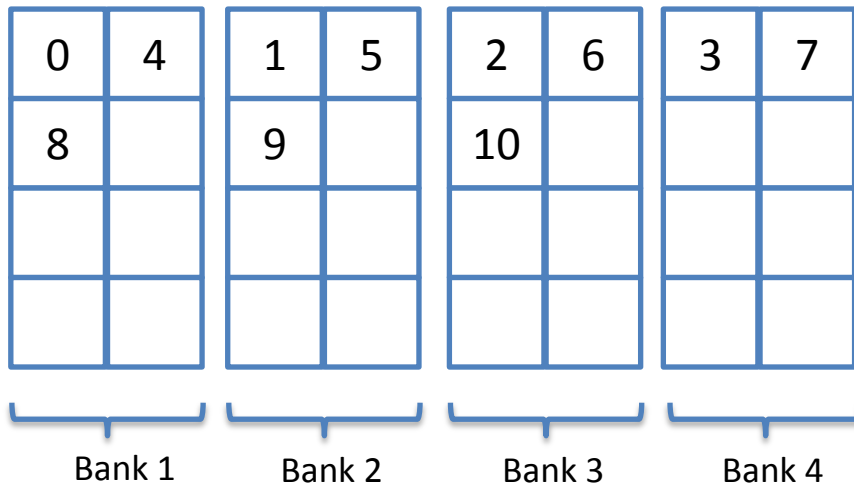


4 level conflict

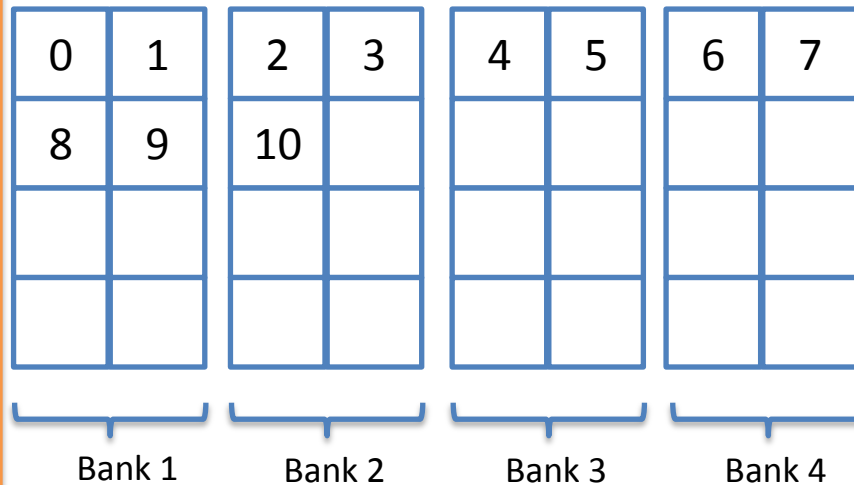


Addressing types

4-byte

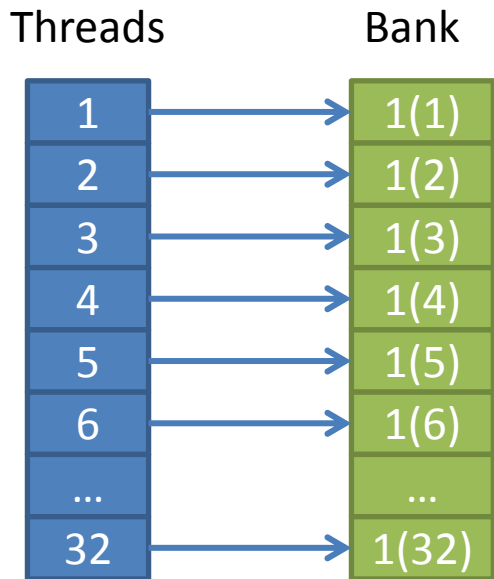


8-byte

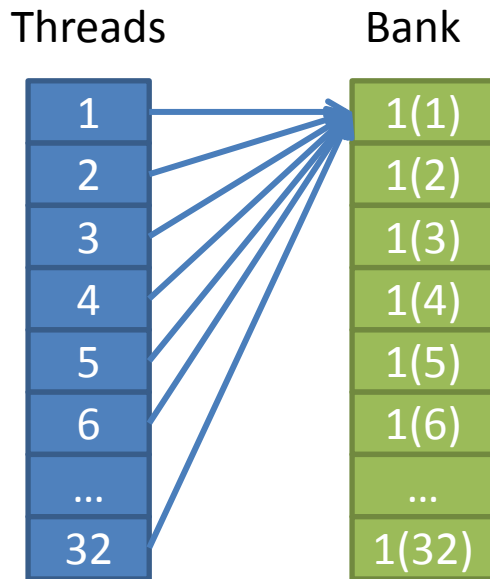




Conflicts



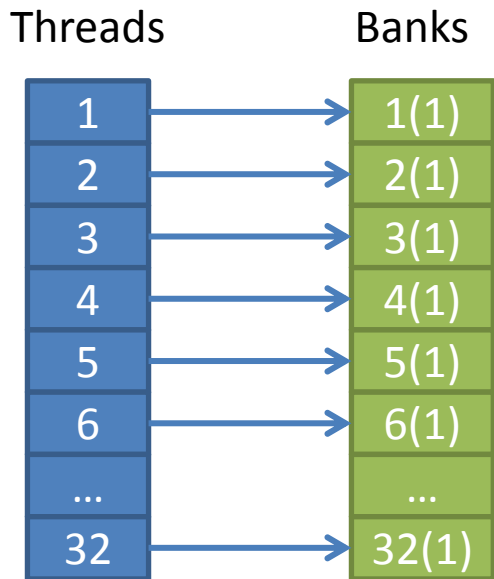
32 Threads request 32 elements from 1 bank.
32 level conflict



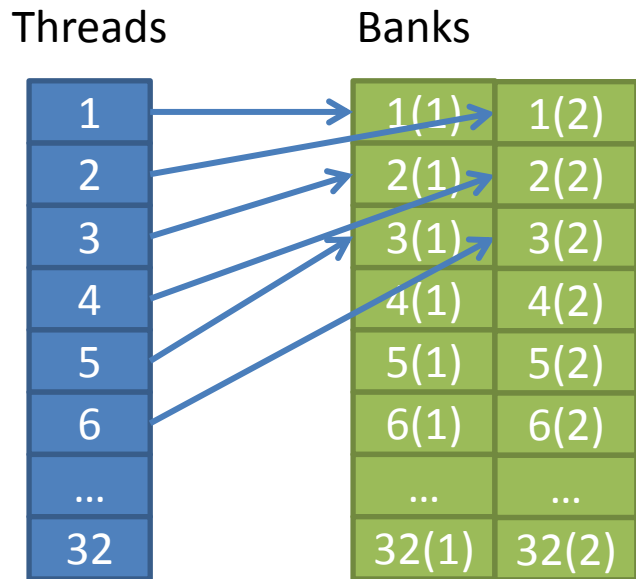
Threads request the same element
Broadcast, no conflicts



Conflicts



32 Threads request 32 8-byte elements
from different Banks



32 Threads request 32
4- byte elements

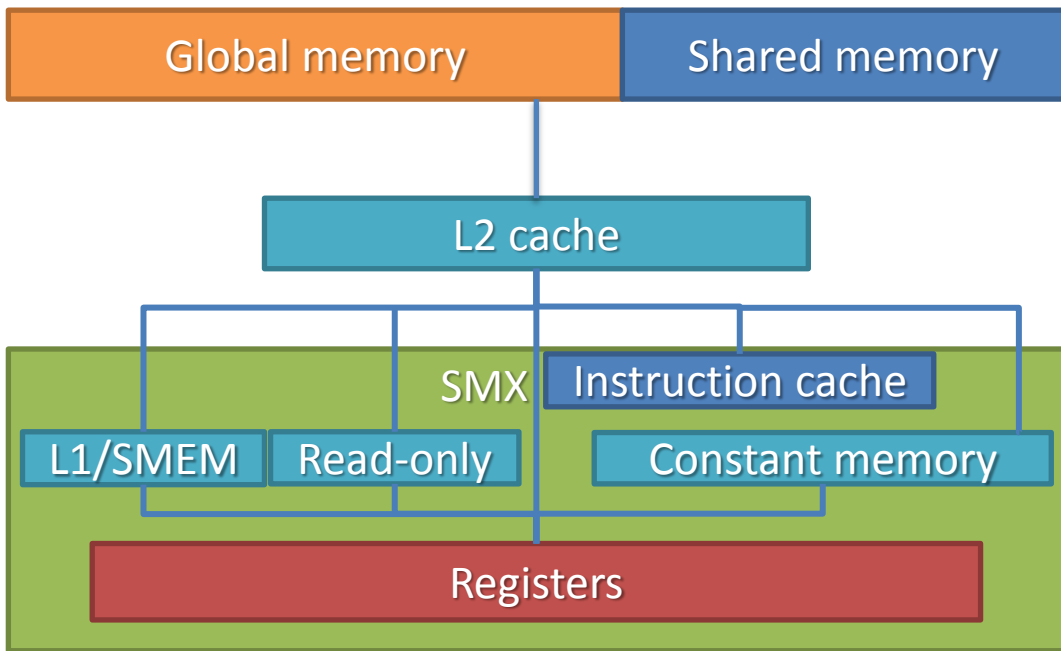
A solid blue banner with a wavy, undulating shape, spanning horizontally across the lower half of the image. It has a slight 3D effect with a thin white shadow on its bottom edge.

Read-only and constant cache



Read-only cache

- 48 KB on SMX separate of L1 cache (origins from texture cache)
- Latency is equal to L1 cache
- Caching 1D, 2D and 3D arrays
- Hardware data filtering, different patterns



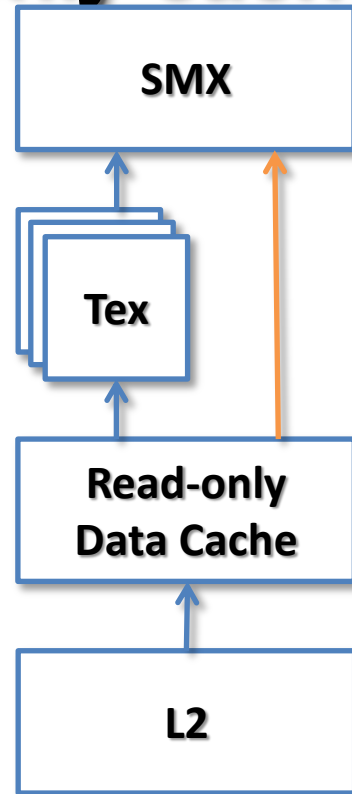


Read-only cache

- `const __restrict`
- Not using texture block
- Cached access to any global address
- Do not need textures preparations

What for?

- Separate of smem/L1 cache
- High bandwidth to L2
- Non-aligned access
- Caching 2-D arrays





const __restrict

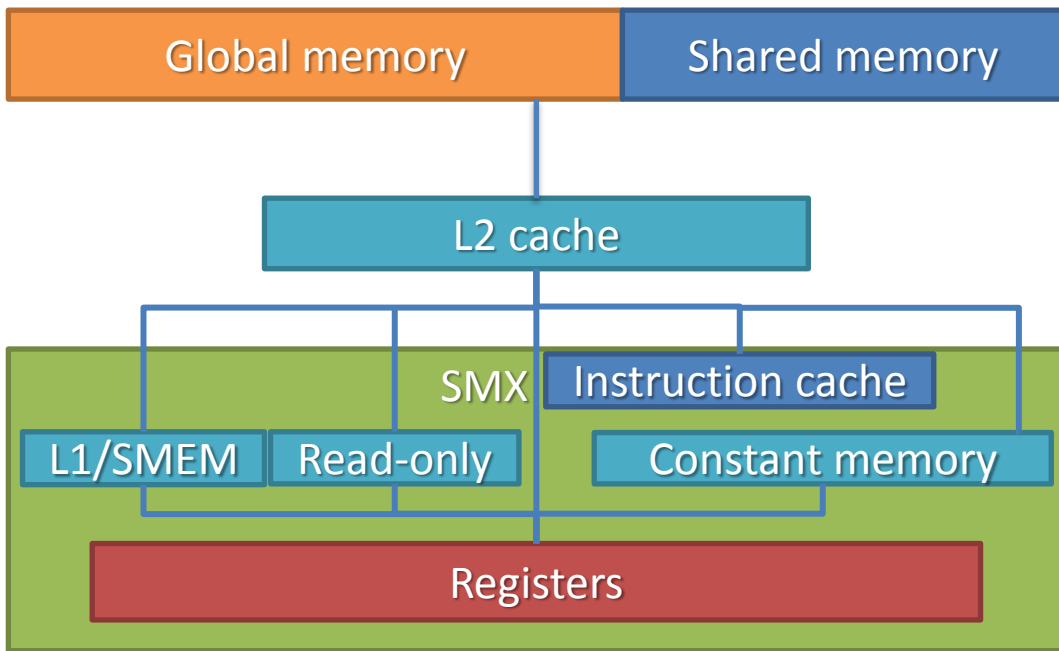
- Specify **const** **__restrict** for parameters
- Compiler generates loading instruction using read-only cache

```
__global__  
void saxpy (float x, float y,  
           const float * __restrict input,  
           float * output){  
    size_t offset = threadIdx.x +  
                    (blockIdx.x *  
                     blockDim.x);  
  
    // Compiler will automatically use  
    // texture  
    // for 'input'  
    output[offset] = (input[offset] * x) + y;  
}
```



Constant memory

- 64 KB separate from L1 and read-only cache
- Access is faster than to global memory
- Accessible only for reading





Constants



constant

- Cached access
- Located in global memory
 - ✓ Max size – 64 KB
 - ✓ Cached with each SMX separately



Can't be changed from kernel

- Doesn't support dynamic arrays
- `cudaMemcpyToSymbol()`
- `cudaMemcpyFromSymbol()`



What for?

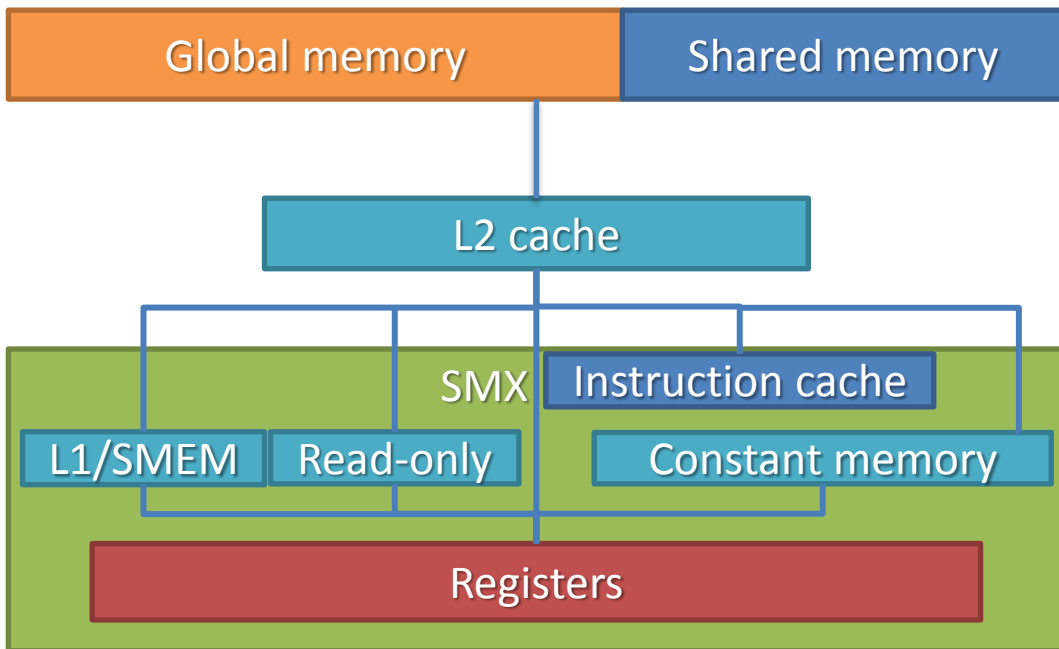
- Separate of smem/L1 and read-only cache
- Latency is less than for L1

Registers and local memory



Registers and local memory

- 65536 32-bit registers for SMX
- The fastest memory, used for computing
- Registers store:
 - Static arrays with predefined size
 - Variables
- Amount of registers allocated per thread defines the amount of blocks issued on SMX concurrently





Registers and local memory

- Compiler controls registers allocation
 - No more than 255 registers per thread
 - Could be artificially limited by `-maxrregcount` (compiler flag)
- If code requires more registers than hardware has there is a register spilling
 - A region of global memory (local memory) imitates registers
 - ✓ L1 and L2 cache requests
 - ✓ High latency

Grid configuration



Grid configuration

Grid – allocation of threads and blocks

- Two main reasons:
 - ✓ Give enough parallel tasks to SMX
 - ✓ Share work between SMXs
- What to consider:
 - ✓ Amount of threads per block
 - ✓ Amount of blocks
 - ✓ Amount of work per block



Device occupancy

- Device occupancy: amount of concurrently issued threads per SMX
 - Could be measured as an amount of threads (warps)
 - Percent of maximum threads per SMX
- Different factors:
 - Amount of registers per thread
 - ✓ All SMX registers are divided between threads
 - Amount of shared memory per block
 - Shared memory is divided between blocks on SMX



Block size

- Block size is multiple of warp size
 - Even if you allocate less threads it will be rounded up by hardware
- Blocks could be too small
 - SMX can run up to 16 blocks concurrently
 - ✓ This can limit performance

SMX resources:

- Registers
- Shared memory



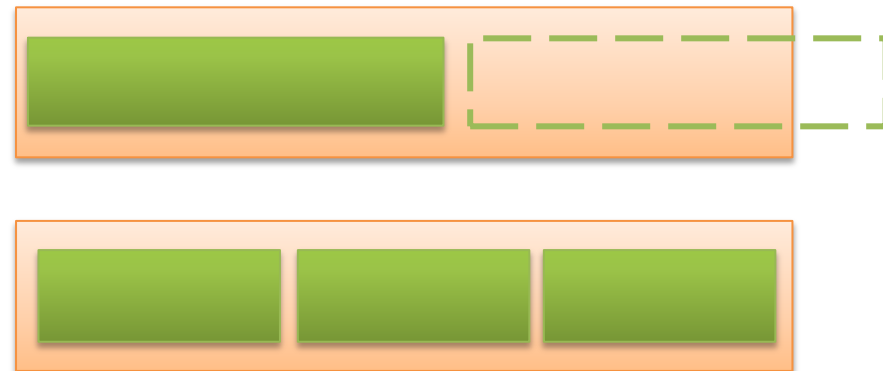


Block size

- Blocks could be too big:
 - There is enough SMX resources to run additional threads but not for one more big block
 - Block starts execution only when there is enough resources for all of its threads

SMX resources:

- Registers
- Shared memory





Occupancy calculator

- ④ In SDK
 - CUDA Samples -> documentation -> CUDA Occupancy Calculator
- ④ Helps to calculate the occupancy using the following parameters:
 - Architecture
 - Configuration of SMEM/L1
 - Threads per block
 - Registers per thread
 - Shared memory per block



Occupancy calculator

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): (Help)
1.b) Select Shared Memory Size Config (bytes)

2.) Enter your resource usage:
Threads Per Block (Help)
Registers Per Thread
Shared Memory Per Block (bytes)

(Don't edit anything below this line)

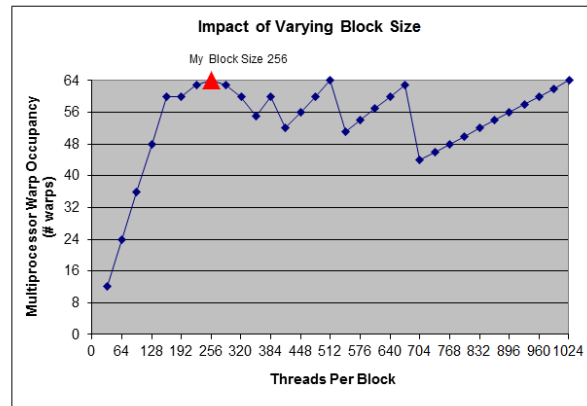
3.) GPU Occupancy Data is displayed here and in the graphs: (Help)
Active Threads per Multiprocessor
Active Warps per Multiprocessor
Active Thread Blocks per Multiprocessor
Occupancy of each Multiprocessor

Physical Limits for GPU Compute Capability:	3.5
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

[Click Here for detailed instructions on how to use this occupancy calculator.](#)

[For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

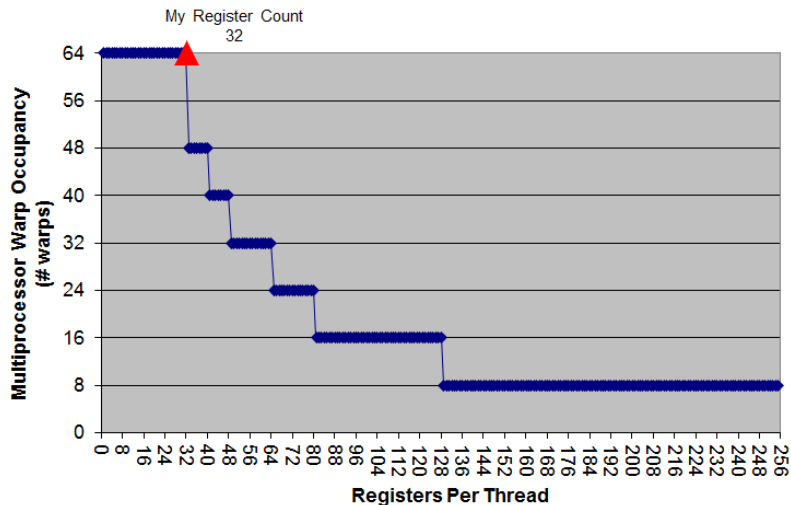
Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



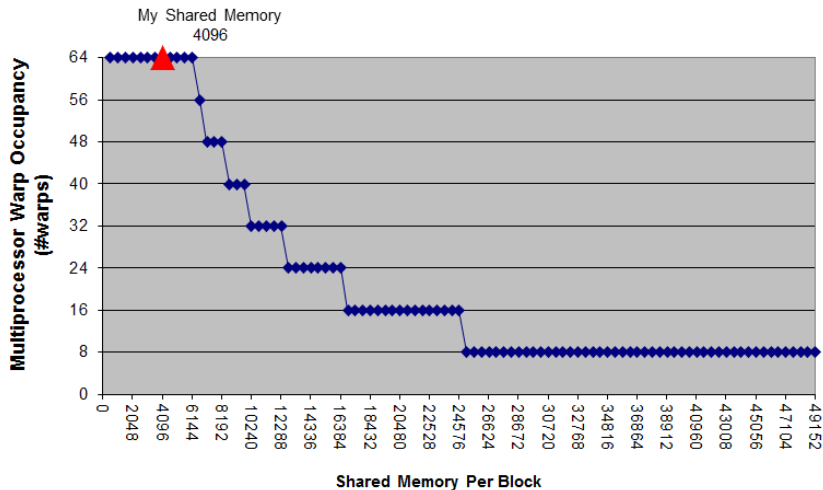


Occupancy calculator

Impact of Varying Register Count Per Thread



Impact of Varying Shared Memory Usage Per Block





Guidelines

Block size:

- Start with 128-256 threads per block
 - ✓ Increase or decrease according to your function parameters
- Multiple of warp size
- If the device occupancy is low:
 - ✓ Check the amount of resources – registers and shared memory

Grid size

- 1000 and more blocks
 - ✓ Uniform distribution of work over GPU
 - ✓ Code will be ready for execution on different architectures

Arithmetic optimizations



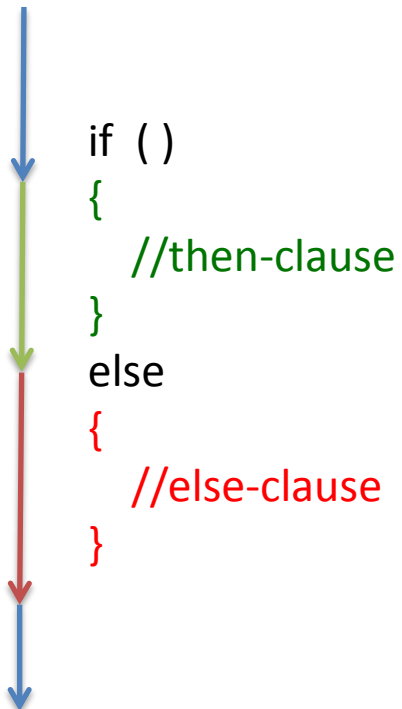

Control flow

- Instructions are issued by warps
 - The entire warp issues the same instruction
- Scheduling
 - Warps are dynamically scheduled
 - Warp scheduler chooses a warp which has instructions ready to run
 - Warp switching helps to hide latency



Control flow

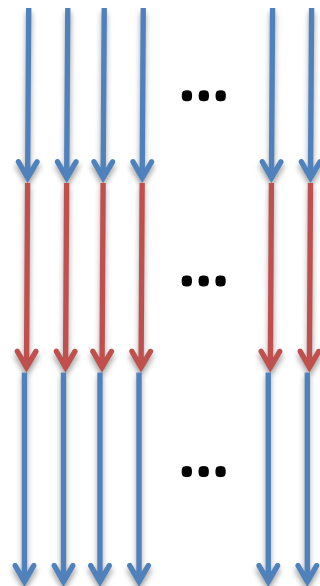
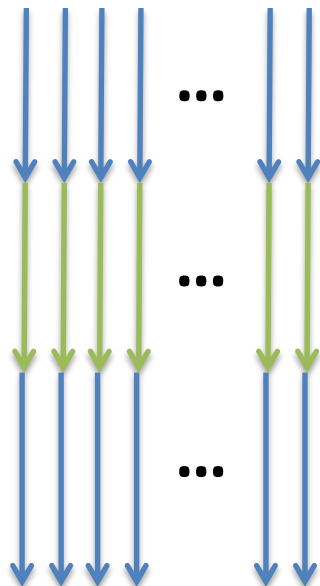

I
n
s
t
r
u
c
t
i
o
n
s





Warp control flow

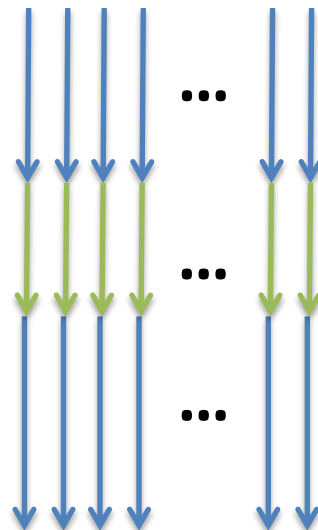
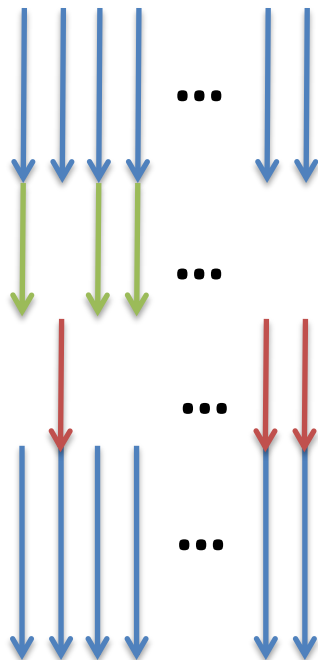
I
n
s
t
r
u
c
t
i
o
n
s





Branching

I
n
s
t
r
u
c
t
i
o
n
s





Instructions

• Performance limiters:

- Maximum amount of instructions per cycle
- Warp branching forces to issue extra instructions

• Serialization

- Warp threads issue the same instruction serial
- Most often cases are:
 - ✓ Shared memory bank conflicts
 - ✓ Uncoalesced memory access



Instructions

- ④ Compare resulting performance with architecture capabilities
 - Profiler measures performance as IPC (instructions per cycle)
 - Peak performance is listed in Programming Guide
- ④ Serialization checking:
 - Amount of repeats: `instructions_issued` – `instructions_executed`
 - Profiler reports
 - ✓ serial running percent – $\text{executed instructions} / \text{issued instructions}$



Instructions

❏ Branching

- Profiler counters: `divergent_branch`, `branch`
- Allows to determine only branching serialization, all other reasons are not counted

❏ Make sure that you use 64-bit arithmetics only where you need

- `fp64` bandwidth is less than `fp32`
- Values without “f” (12.2 instead of 12.2f) interpreted as `fp64`

A blue wavy banner with the word 'Intrinsics' in white text.

Intrinsics



Intrinsics

- Set and performance depends on the architecture
- Usually a sequence of software instructions can be replaced with one hardware instruction with less tolerance



Instructions

- Use intrinsics if it's possible (`__sin()`, `__sincos()`, `__exp()`)
 - Available for many math functions
 - Tolerance is 2-3 bits less but performance is better.



Performance test

Test structure

- Generating 2 sets of 100000000 random values
- Measuring time for 2 kernels:
 - ✓ Usual instruction
 - ✓ Intrinsic
- Output
 - ✓ Time
 - ✓ First 10 values to compare tolerance



Example

```
__global__ void coss (float *x,float  
                    *y,float *r,int n){  
    int indexX = blockIdx.x *  
                (blockDim.x) + threadIdx.x;  
    if (indexX<n){  
        float a,b,c;  
        b=x[indexX];  
        c=y[indexX];  
        #pragma unroll  
        for (int i=0; i<1000;i++)  
            a+=cosf(b)+cosf(c);  
        r[indexX]=a;  
    }  
}
```

```
__global__ void coss_i(float *x,float  
                    *y,float *r,int n){  
    int indexX = blockIdx.x *  
                (blockDim.x) + threadIdx.x;  
    if (indexX<n){  
        float a,b,c;  
        b=x[indexX];  
        c=y[indexX];  
        #pragma unroll  
        for (int i=0; i<1000;i++)  
            a+=__cosf(b)+__cosf(c);  
        r[indexX]=a;  
    }  
}
```



Results

tolerance (for cos)

GPU res	Intrinsic GPU res
1697,11	1697,85
1485,11	1485,71
1683,31	1684,31
1346,15	1346,71
1453,87	1454,47
1833,71	1834,61
1383,51	1384,29
1923,46	1924,46
1836,79	1837,68
1842,56	1843,43

Time

	GPU time	Intrinsic GPU time
cos	1372,43	21,92
sin	1298,64	21,92
exp10	22,61	22,05
exp	22,65	22,05
fadd	21,85	21,84
fdiv	22,49	22,48
fmaf	21,84	21,84
powf	3,32	1,10
fsqrt	3,00	3,00



Compiler flags

Additional compiler flags:

- `-ftz=true`: replacing subnormal values with 0
- `-prec-div=false` : fast division instruction (some precision losses)
- `-prec-sqrt=false` : fast square root calculation (some precision losses)
- `-use_fast_math` : use intrinsics

Example



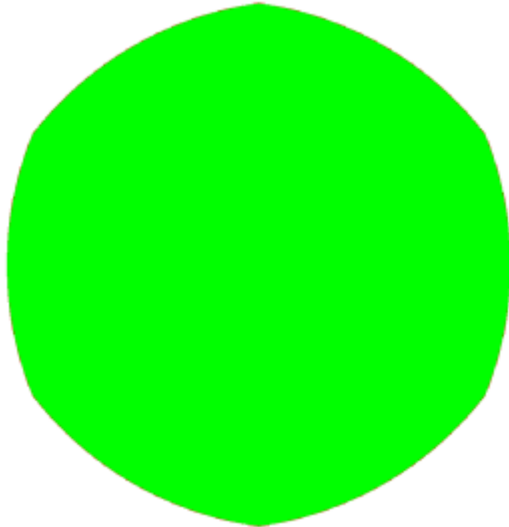
Stewart platform



- Scanning Stewart platform workspace
- 350 slices by 1000×1000 points
- Uses `asinf()`



Output data



- HexapodSliceKernel generates bmp images of slices
- HexapodKernel calculates the circumcircle radius



1: Block size

Scanning workspace:

- HexapodSliceKernel:
 - ✓ 27 registers per thread, **3977** bytes SMEM per warpна
 - ✓ No more than **12** warps per SM

Initial grid configuration: 16×16 threads per block, 63×63 blocks, **350** kernel call

- Blocks are too big – **1 SM** holds only **1 блок**
- Device occupancy– **16%**
- Time: 5.6 s.



2: Block size

Optimized:

- **10*10** threads pre block, **100*100*350** blocks
- **12472** bytes SMEM per block
- Kernel and copy called only once
- Occupancy– **25%**
- Time– 2,4 s.

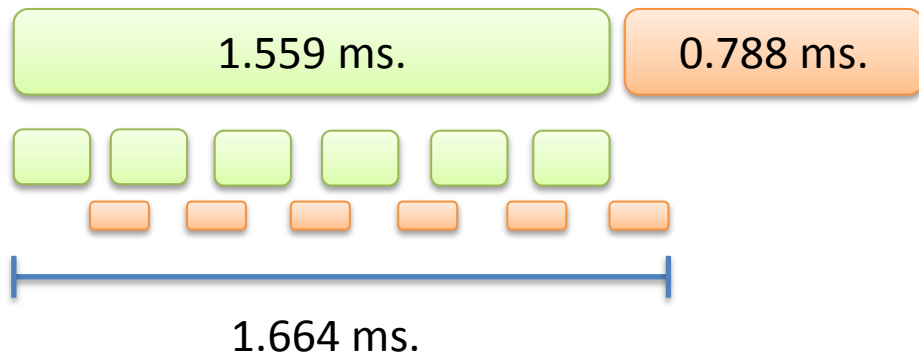
Not finished yet!



3: Asynchrony

Asynchronous copy

- Split kernel into several iterations
- `cudaMallocHost()`
- `cudaMemcpyAsync()`
- `cudaDeviceSynchronize()`





4: Block size

Optimized:

- **16*16** threads per block **63*63*50** blocks.
- **72** SMEM per block, **29** registers.
- Kernel and copy are called **7** times .
- Occupancy– **66,7%**
- Time– 1.096 s.
- Speedup– **5.2** times.



● Instruction Level Parallelism:

- **Each thread** calculates **2** output **elements**
- **8*16** threads per block **63*63*50** blocks.
- **0** bytes SMEM per block, **47** registers.
- Kernel and copy called **7** times.
- Occupancy– **41,7%(83,4%)**
- Time– 1.090 s.



ILP (adding **#pragma unroll**):

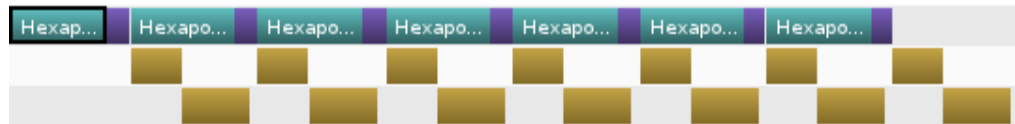
- **Each thread** calculates **2** output **elements**
- **8*16** threads per block **63*63*50** blocks.
- **0** bytes SMEM per block, **47** registers.
- Kernel and copy are called **7** times.
- Occupancy – **41,7%(83,4%)**
- Time– 1,023 s.



7:Fast math

Using intrinsics:

- **8*16** per block
63*63*50 blocks.
- **0** bytes SMEM per block, **40** registers.
- Kernel and copy are called **7** times.
- Occupancy – **50%(100%)**
- Time– **0,433** s.

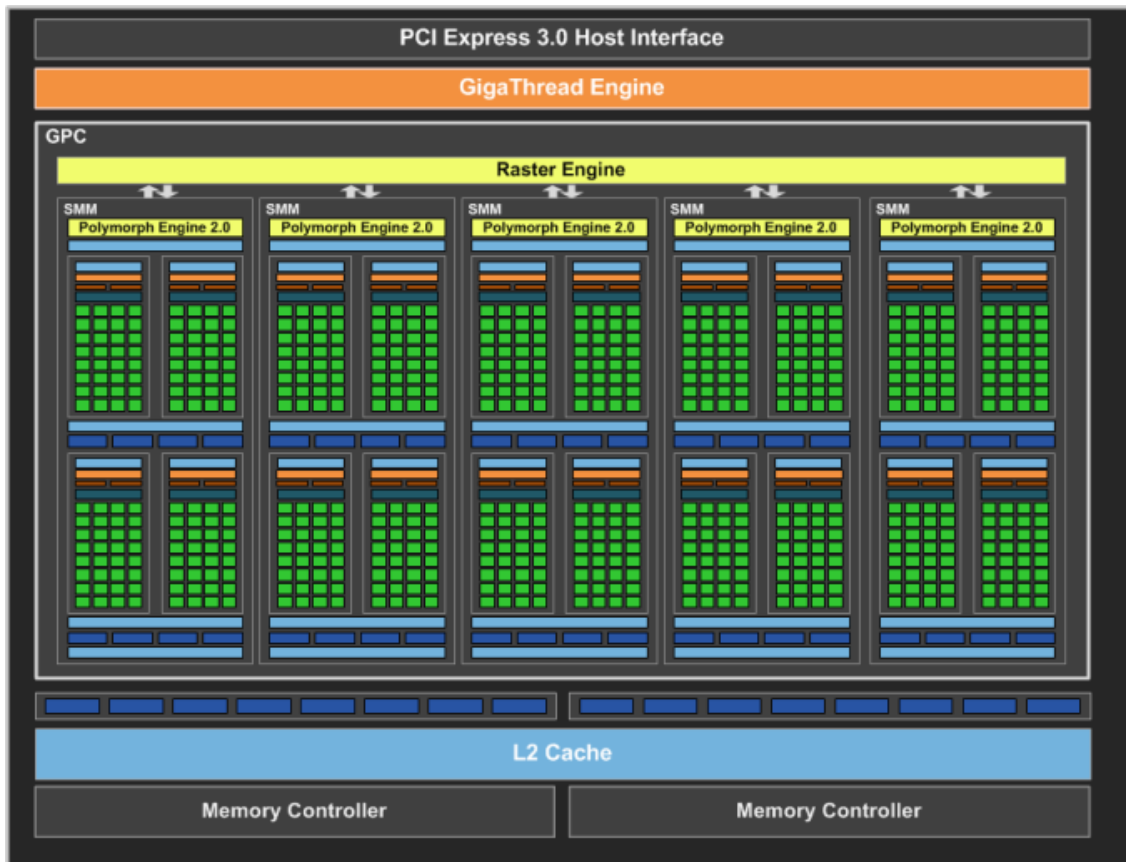


Time is 12,9 times less!!!



Maxwell architecture

- 5 SMM
- 640 CUDA cores
- PCI-e 3.0
- 2 MB L2 cache





SMM architecture

- 128 CUDA cores
- 8 texture processors
- 65536 32-byte registers
- 4 warp schedulers
- 64 KB L1/ texture cache
- 64 KB Shared memory
- 32 LD/ST unit
- 32 SF unit





Results

- GPU architecture
 - Kepler
 - Maxwell
- Performance limiters
- Memory
 - Global memory
 - Shared memory
 - Read-only and constant cache
 - Registers and local memory
- Grid configuration
- Arithmetics
 - Branching
 - Intrinsics
- Asynchrony
- Case study