

# CS 380 - GPU and GPGPU Programming

## Lecture 22: Stream Computing and GPGPU; CUDA Memory, Pt. 1

Markus Hadwiger, KAUST



# Reading Assignment #12 (until Nov 20)

## Read (required):

- Programming Massively Parallel Processors book, 4<sup>th</sup> edition

**Chapter 5** (Memory architecture and data locality)

**Chapter 6** (Performance considerations)

## Read (optional):

- Stream processing

[https://en.wikipedia.org/wiki/Stream\\_processing](https://en.wikipedia.org/wiki/Stream_processing)

- Linear algebra operators for GPU implementation of numerical algorithms,  
Krueger and Westermann, SIGGRAPH 2003

<https://dl.acm.org/doi/10.1145/882262.882363>

- A Survey of General-Purpose Computation on Graphics Hardware (2007)

<https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x>



# Next Lectures

Lecture 23: Sun, Nov 20

Lecture 24: Tue, Nov 22 (make-up lecture; 16:00 – 17:15)

Lecture 25: Wed, Nov 23

# Stream Computing and GPGPU



# Types of Parallelism

## Bit-Level Parallelism (70s and 80s)

- Doubling the word size 4, 8, 16, 32-bit (64-bit ~2003)

## Instruction-Level Parallelism (mid 80s-90s)

- Instructions are split into stages → multi stage pipeline
- Superscalar execution, ...



## Data Parallelism

- Multiple processors execute the same instructions on different parts of the data

## Task Parallelism

- Multiple processors execute instructions independently



# From GPU to GPGPU

1990s Fixed function graphics-pipeline used for more general computations in academia (e.g., rasterization, z-buffer)

2001 Shaders changed the API to access graphics cards

2004 Brook for GPUs changed the terminology

Since then:

ATI's Stream SDK (originally based on Brook)

NVIDIA's CUDA (started by Brook developers)

OpenCL (platform independent)

GLSL Compute Shaders (platform independent)

Vulkan Compute Shaders (platform independent)

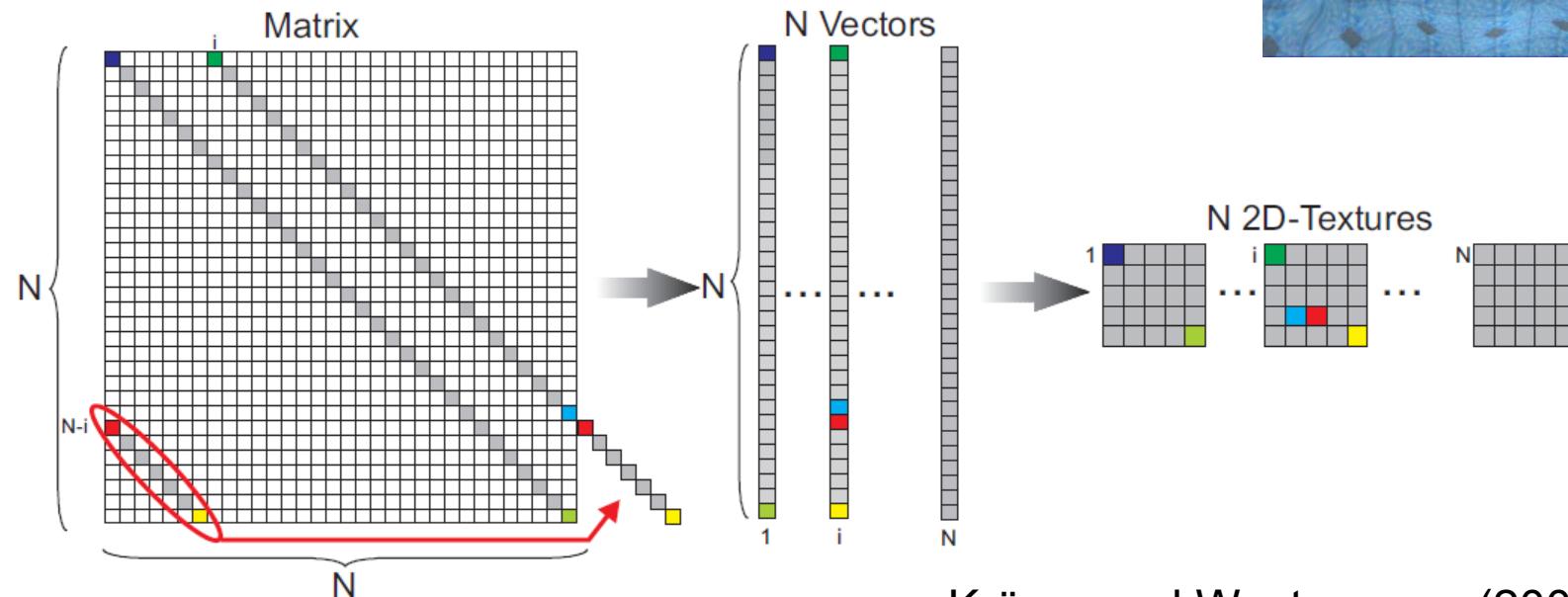
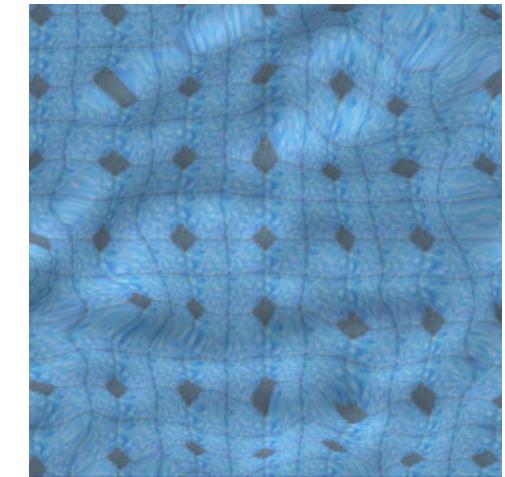
DirectX 12 Compute Shaders

# Early GPGPU: Linear Algebra Operators



Vector and matrix representation and operators

- Early approach based on graphics primitives
- Now CUDA makes this much easier
- Linear systems solvers



Krüger and Westermann (2003)

# Stream Programming Abstraction



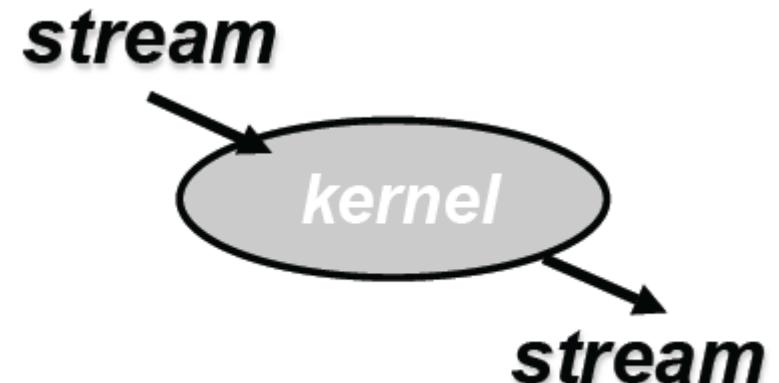
Goal: SW programming model that matches data parallelism

## Streams

- Collection of data records
- All data is expressed in streams

## Kernels

- Inputs/outputs are streams
- Perform computation on streams  
(each data record is processes independently)
- Can be chained together



Courtesy John Owens



# Why Streams?

- Exposing parallelism
  - Data parallelism
  - Task parallelism

```
for(i = 0; i<size; i++)  
{  
    a[i] = 2*b[i];  
}
```

```
for(each a, b)  
{  
    a = 2*b;  
}
```

```
for(i = 0; i<size; i++)  
{  
    a[i] = a[i+1]*2;  
}
```

```
for(each a)  
{  
    ???  
}
```

- Multiple stream elements can be processed in parallel
- Multiple tasks can be processed in parallel
- Predictable memory access pattern
- Optimize for throughput of all elements, not latency of one
- Processing many elements at once allows latency hiding

# Brook for GPUs: Stream Computing on Graphics Hardware

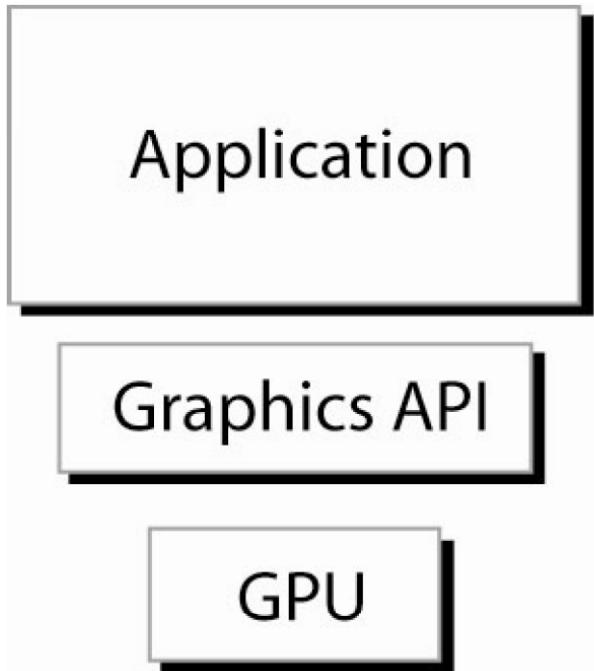


Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan

Computer Science Department  
Stanford University

# domain specific solutions

---

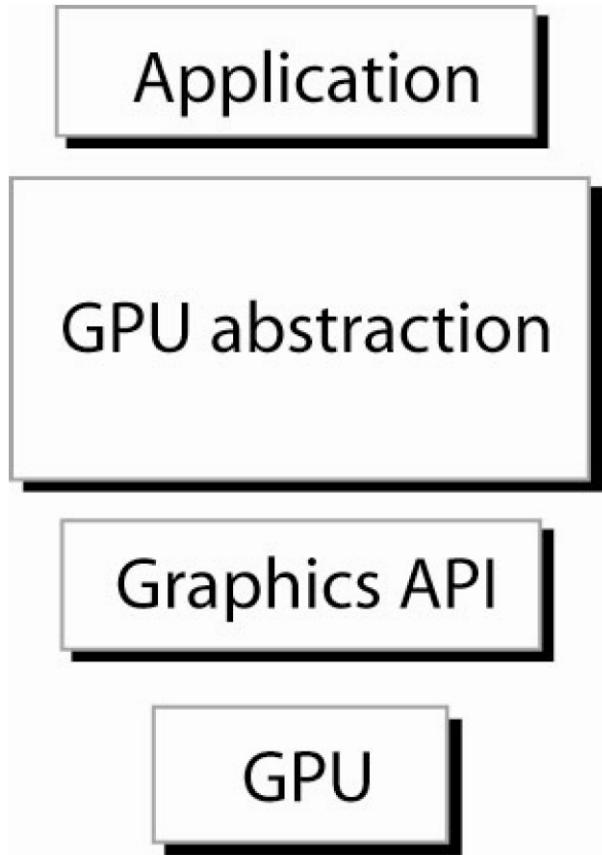


map directly to graphics primitives

requires extensive knowledge of GPU programming

# building an abstraction

---



general GPU computing question

- can we simplify GPU programming?
- what is the correct abstraction for GPU-based computing?
- what is the scope of problems that can be implemented efficiently on the GPU?

# contributions

---



- Brook stream programming environment for GPU-based computing
  - language, compiler, and runtime system
- virtualizing or extending GPU resources
- analysis of when GPUs outperform CPUs

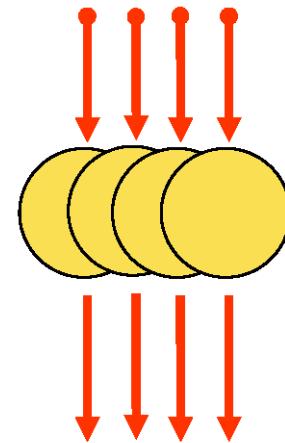
# GPU programming model

---



each fragment shaded independently

- no dependencies between fragments
  - temporary registers are zeroed
  - no static variables
  - no read-modify-write textures
- multiple “pixel pipes”



# GPU = data parallel

---



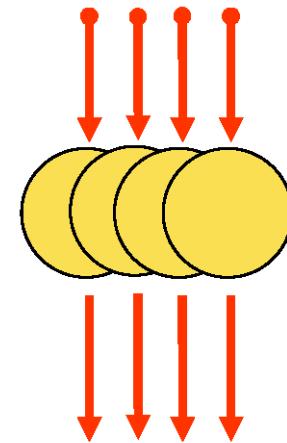
each fragment shaded independently

- no dependencies between fragments
  - temporary registers are zeroed
  - no static variables
  - no read-modify-write textures
- multiple “pixel pipes”

## **data parallelism**

- support ALU heavy architectures
- hide memory latency

[Torborg and Kajiya 96, Anderson et al. 97, Igehy et al. 98]



# Brook language

---



## stream programming model

- enforce data parallel computing
  - streams
- encourage arithmetic intensity
  - kernels

# design goals

---



- general purpose computing  
    GPU = general streaming-coprocessor
- GPU-based computing for the masses  
    no graphics experience required  
    eliminating annoying GPU limitations
- performance
- platform independent  
    ATI & NVIDIA  
    DirectX & OpenGL  
    Windows & Linux

# Brook language

---



## C with streams

- streams
  - collection of records requiring similar computation
    - particle positions, voxels, FEM cell, ...

```
Ray r<200>;  
float3 velocityfield<100,100,100>;
```

- data parallelism
  - provides data to operate on in parallel



- kernels
  - functions applied to streams

- similar to for\_all construct

```
kernel void foo (float a<0>, float b<0>,
                 out float result<0>) {
    result = a + b;
}
```

```
float a<100>;
float b<100>;
float c<100>;
```

```
foo(a,b,c);
```

```
for (i=0; i<100; i++)
    c[i] = a[i]+b[i];
```



# kernels

---



- kernels arguments
  - input/output streams

```
kernel void foo (float a<>,
                float b<>,
                out float result<>) {
    result = a + b;
}
```

# kernels

---



- kernels arguments
  - input/output streams
  - gather streams

```
kernel void foo (..., float array[] ) {  
    a = array[i];  
}
```

# kernels

---



- kernels arguments
  - input/output streams
  - gather streams
  - iterator streams

```
kernel void foo (... , iter float n<> ) {  
    a = n + b;  
}
```

# kernels

---



- kernels arguments
  - input/output streams
  - gather streams
  - iterator streams
  - constant parameters

```
kernel void foo (..., float c) {  
    a = c + b;  
}
```

# *Brook language* kernels



- Ray Triangle Intersection

```
kernel void krnIntersectTriangle(Ray ray<in>, Triangle tris[],
                                 RayState oldraystate<in>,
                                 GridTrilist trilist[],
                                 out Hit candidatehit<out>) {
    float idx, det, inv_det;
    float3 edge1, edge2, pvec, tvec, qvec;
    if(oldraystate.state.y > 0) {
        idx = trilist[oldraystate.state.w].trinum;
        edge1 = tris[idx].v1 - tris[idx].v0;
        edge2 = tris[idx].v2 - tris[idx].v0;
        pvec = cross(ray.d, edge2);
        det = dot(edge1, pvec);
        inv_det = 1.0f/det;
        tvec = ray.o - tris[idx].v0;
        candidatehit.data.y = dot(tvec, pvec) * inv_det;
        qvec = cross(tvec, edge1);
        candidatehit.data.z = dot(ray.d, qvec) * inv_det;
        candidatehit.data.x = dot(edge2, qvec) * inv_det;
        candidatehit.data.w = idx;
    } else {
        candidatehit.data = float4(0,0,0,-1);
    }
}
```

# reductions

---



- reductions
  - compute single value from a stream

```
reduce void sum (float a<>,
                 reduce float r<>)
    r += a;
}
```

# reductions

---



- reductions
  - compute single value from a stream

```
reduce void sum (float a<>,
                 reduce float r<>)
    r += a;
}
```

```
float a<100>;
float r;
```

```
sum(a,r);
```

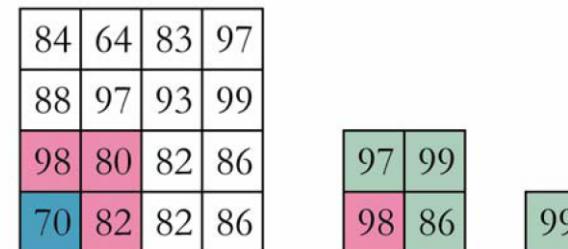
```
r = a[0];
for (int i=1; i<100; i++)
    r += a[i];
```

# reductions



- reductions
  - associative operations only
$$(a+b)+c = a+(b+c)$$
    - sum, multiply, max, min, OR, AND, XOR
    - matrix multiply
  - permits parallel execution

31	41	59	26	53	58	97	93
23	84	62	64	33	83	27	95
2	88	41	97	16	93	99	37
51	5	82	9	74	83	94	45
92	30	78	16	40	62	86	20
89	98	62	80	34	82	53	42
11	70	6	79	82	14	80	86
51	32	82	30	66	47	9	38



# *Brook language* reductions

---



- multi-dimension reductions
  - stream “shape” differences resolved by reduce function

# *Brook language* reductions

---



- multi-dimension reductions
  - stream “shape” differences resolved by reduce function

```
reduce void sum (float a<>,
               reduce float r<>)
{
    r += a;
}

float a<20>;
float r<5>;

sum(a,r);
```

# *Brook language* reductions



- multi-dimension reductions
  - stream “shape” differences resolved by reduce function

```
reduce void sum (float a<>,
                 reduce float r<>)
    r += a;
}
```

```
float a<20>;
float r<5>;
```

```
sum(a,r);
```

```
for (int i=0; i<5; i++)
    r[i] = a[i*4];
    for (int j=1; j<4; j++)
        r[i] += a[i*4 + j];
```

# *Brook language* reductions



- multi-dimension reductions
  - stream “shape” differences resolved by reduce function

```
reduce void sum (float a<>,
              reduce float r<>)
    r += a;
}
```

```
float a<20>;
float r<5>;
```

The diagram shows a 5x4 grid of colored squares. The top row is yellow, the second row is orange, the third row is red, the fourth row is brown, and the bottom row is magenta. This visualizes how a 20-element array (a) is mapped onto a 5-element array (r) through a stride of 4.

```
sum(a,r);
```



```
for (int i=0; i<5; i++)
    r[i] = a[i*4];
    for (int j=1; j<4; j++)
        r[i] += a[i*4 + j];
```

## stream repeat & stride



- kernel arguments of different shape
  - resolved by repeat and stride



## stream repeat & stride

- kernel arguments of different shape
  - resolved by repeat and stride

```
kernel void foo (float a<>, float b<>,  
                 out float result<>);
```

```
float a<20>;  
float b<5>;  
float c<10>;
```

```
foo(a,b,c);
```



# stream repeat & stride

- kernel arguments of different shape
  - resolved by repeat and stride

```
kernel void foo (float a<>, float b<>,
                 out float result<>);
```

```
float a<20>;
float b<5>;
float c<10>;
foo(a,b,c);
```

```
foo(a[0],  b[0],  c[0])
foo(a[2],  b[0],  c[1])
foo(a[4],  b[1],  c[2])
foo(a[6],  b[1],  c[3])
foo(a[8],  b[2],  c[4])
foo(a[10], b[2],  c[5])
foo(a[12], b[3],  c[6])
foo(a[14], b[3],  c[7])
foo(a[16], b[4],  c[8])
foo(a[18], b[4],  c[9])
```



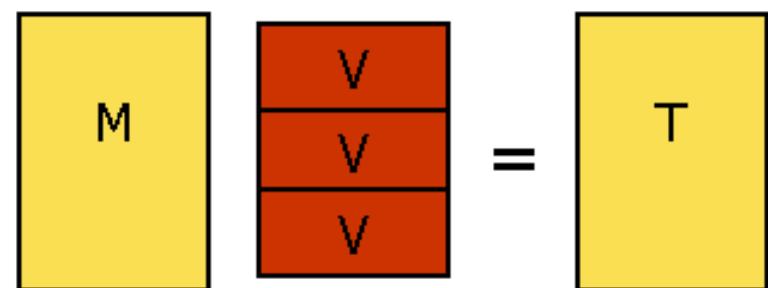
# matrix vector multiply

```
kernel void mul (float a<>, float b<>,
                  out float result<>) {
    result = a*b;
}

reduce void sum (float a<>,
                  reduce float result<>) {
    result += a;
}

float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);
```





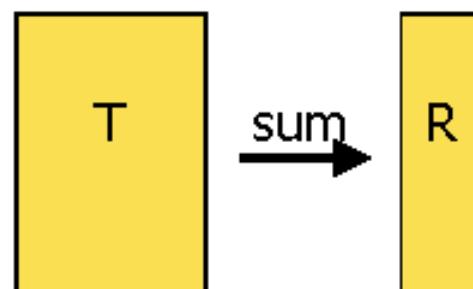
# matrix vector multiply

```
kernel void mul (float a<>, float b<>,
                 out float result<>) {
    result = a*b;
}

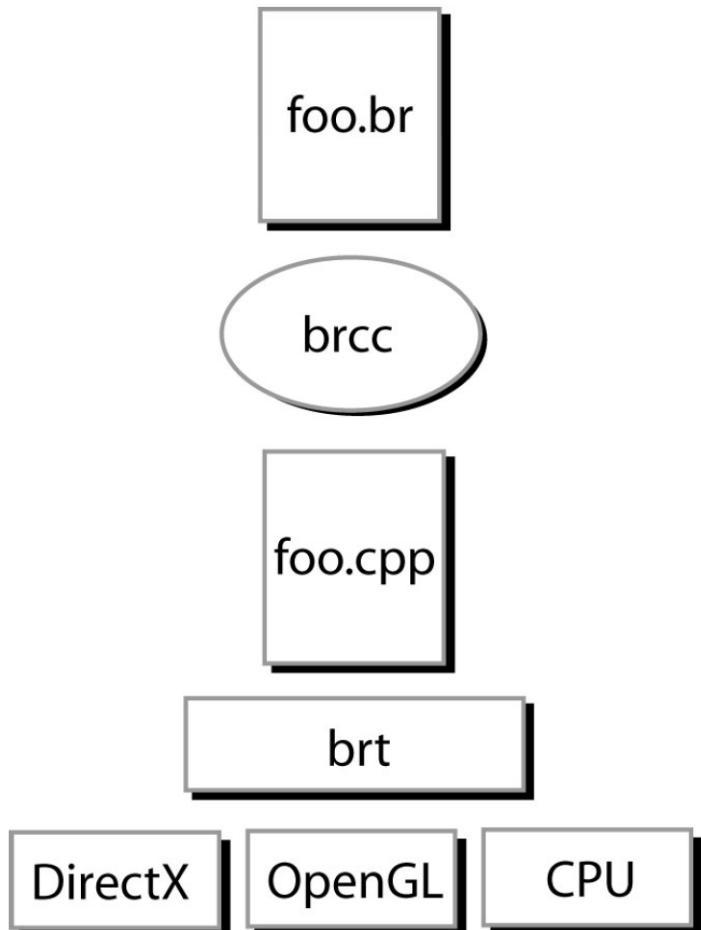
reduce void sum (float a<>,
                 reduce float result<>) {
    result += a;
}

float matrix<20,10>;
float vector<1, 10>;
float tempmv<20,10>;
float result<20, 1>;

mul(matrix,vector,tempmv);
sum(tempmv,result);
```



# system outline



## brcc

- source to source compiler
  - generate CG & HLSL code
  - CGC and FXC for shader assembly
  - virtualization

## brt

- Brook run-time library
  - stream texture management
  - kernel shader execution

# eliminating GPU limitations

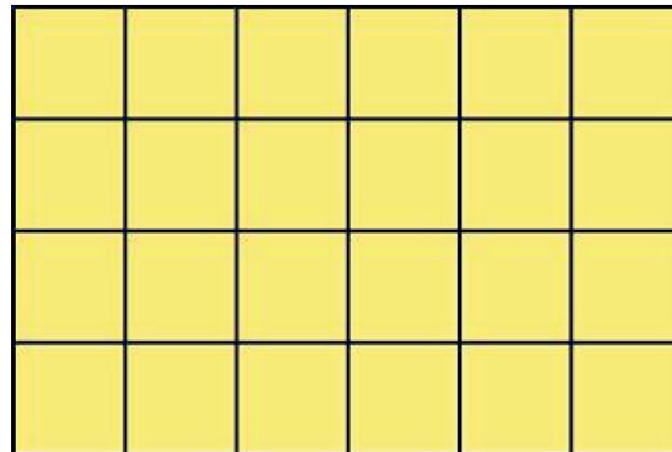
---



treating texture as memory

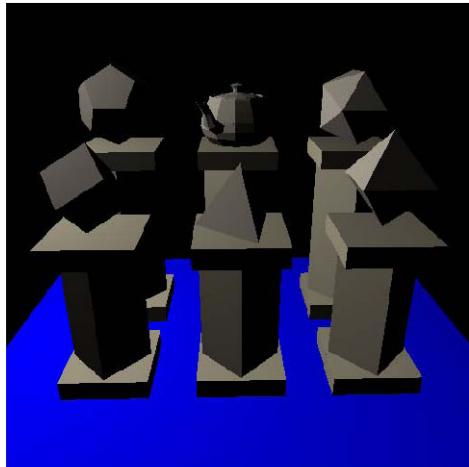
- limited texture size and dimension
- compiler inserts address translation code

```
float matrix<8096,10,30,5>;
```

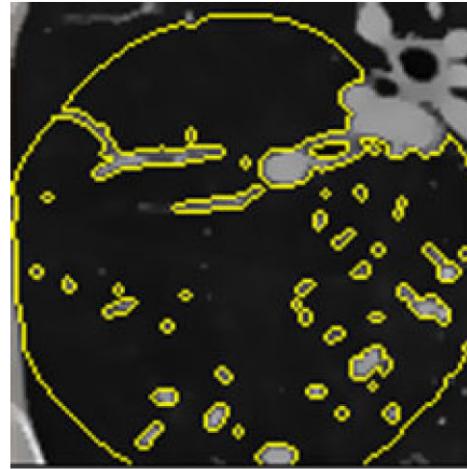


# applications

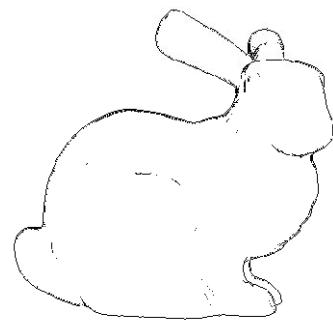
---



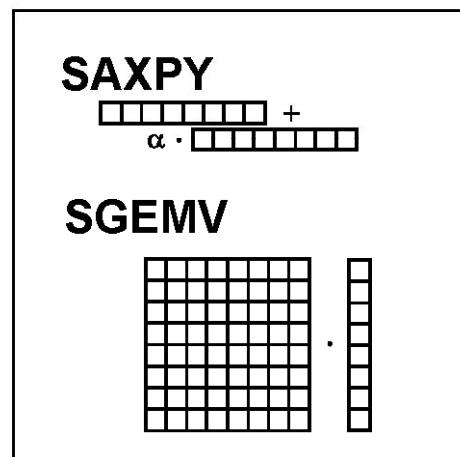
ray-tracer



segmentation



fft edge detect



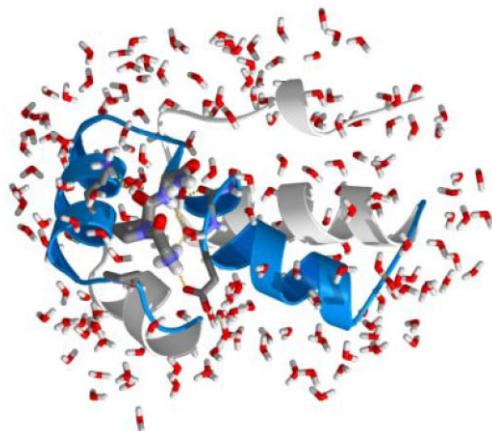
linear algebra

# summary

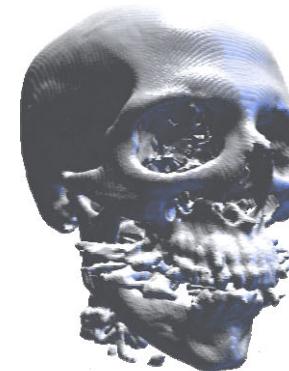
---



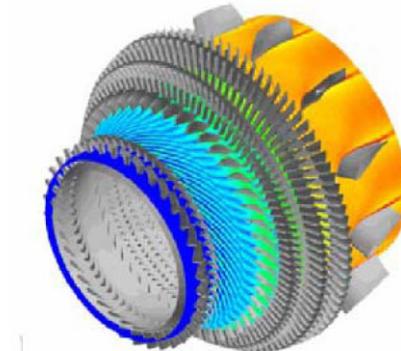
## GPU-based computing for the masses



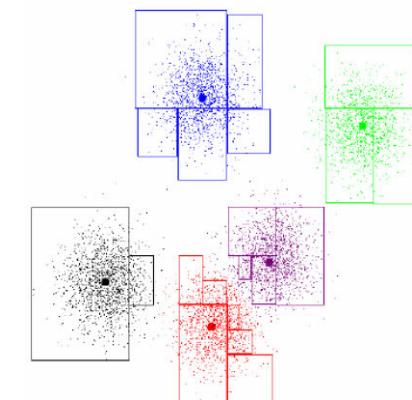
bioinfomatics



rendering



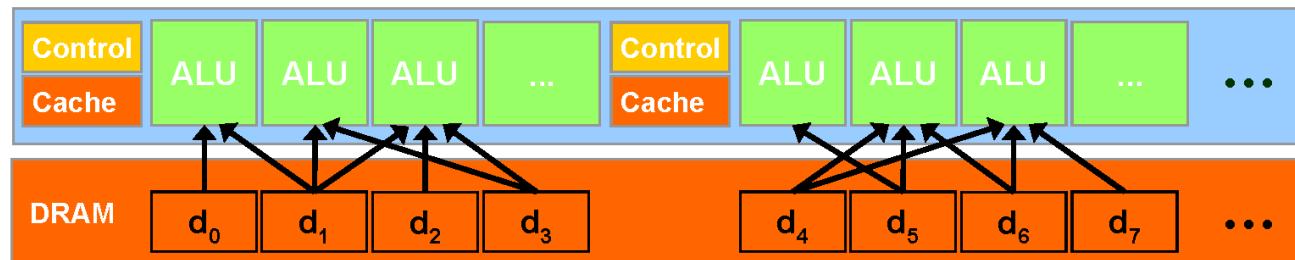
simulation



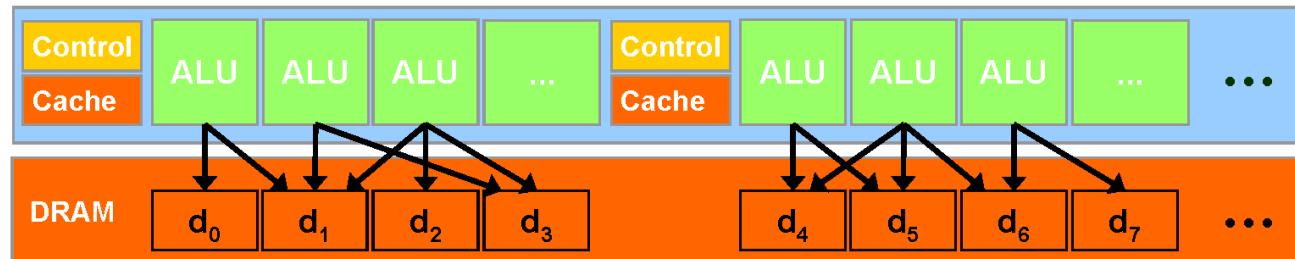
statistics

# CUDA Highlights: Scatter

- CUDA provides generic DRAM memory addressing
  - Gather:



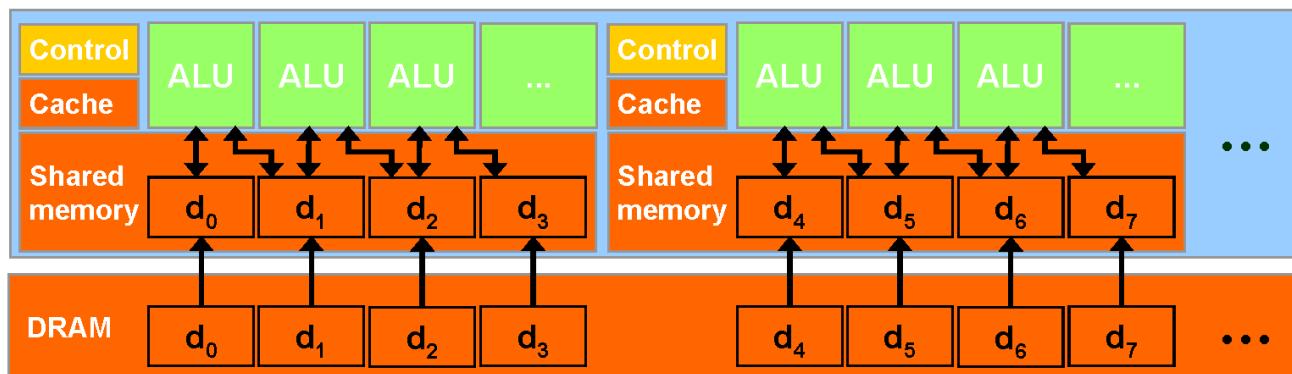
- And scatter: no longer limited to write one pixel



→ More programming flexibility

# CUDA Highlights: On-Chip Shared Memory

- CUDA enables access to a parallel **on-chip shared memory** for efficient inter-thread data sharing

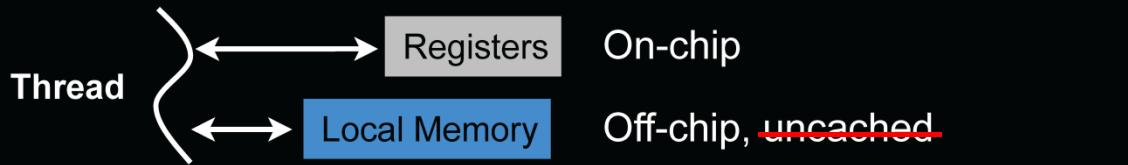


→ **Big memory bandwidth savings**

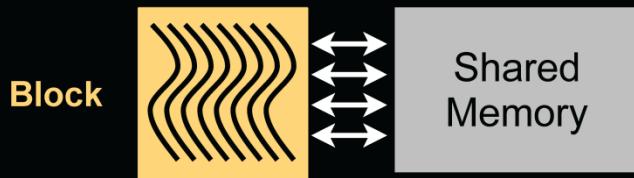
# CUDA Memory: Overview

# Kernel Memory Access

- Per-thread

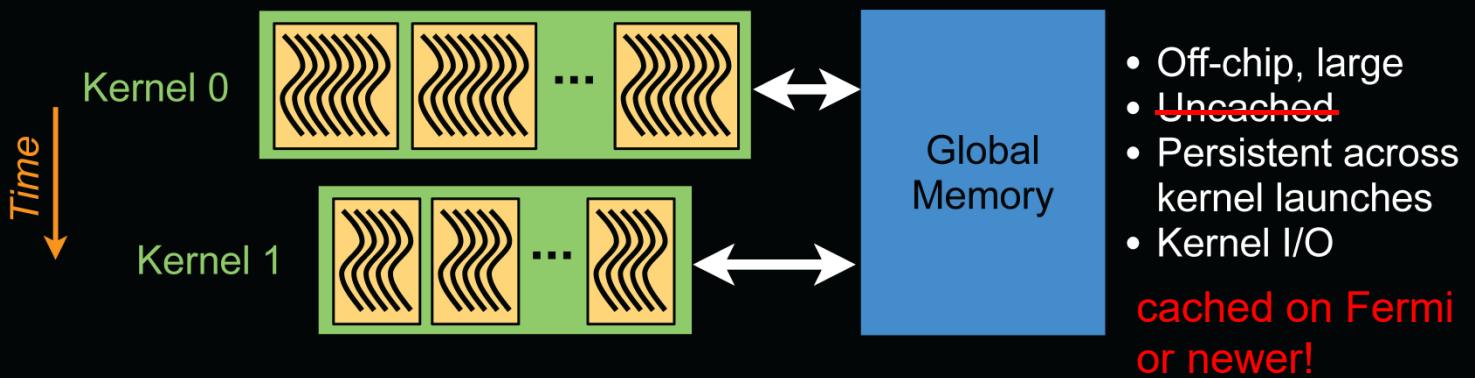


- Per-block



- On-chip, small
- Fast

- Per-device





# Memory and Cache Types

## Global memory

- [Device] **L2 cache**
- [SM] **L1 cache** (shared mem carved out; or L1 shared with tex cache)
- [SM/TPC] **Texture cache** (separate, or shared with L1 cache)
- [SM] **Read-only data cache** (storage might be same as tex cache)

## Shared memory

- [SM] Shareable only between threads in same thread block  
(Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing



# **Memory Configurations and Types for Different Compute Capabilities**

# GK104 SMX

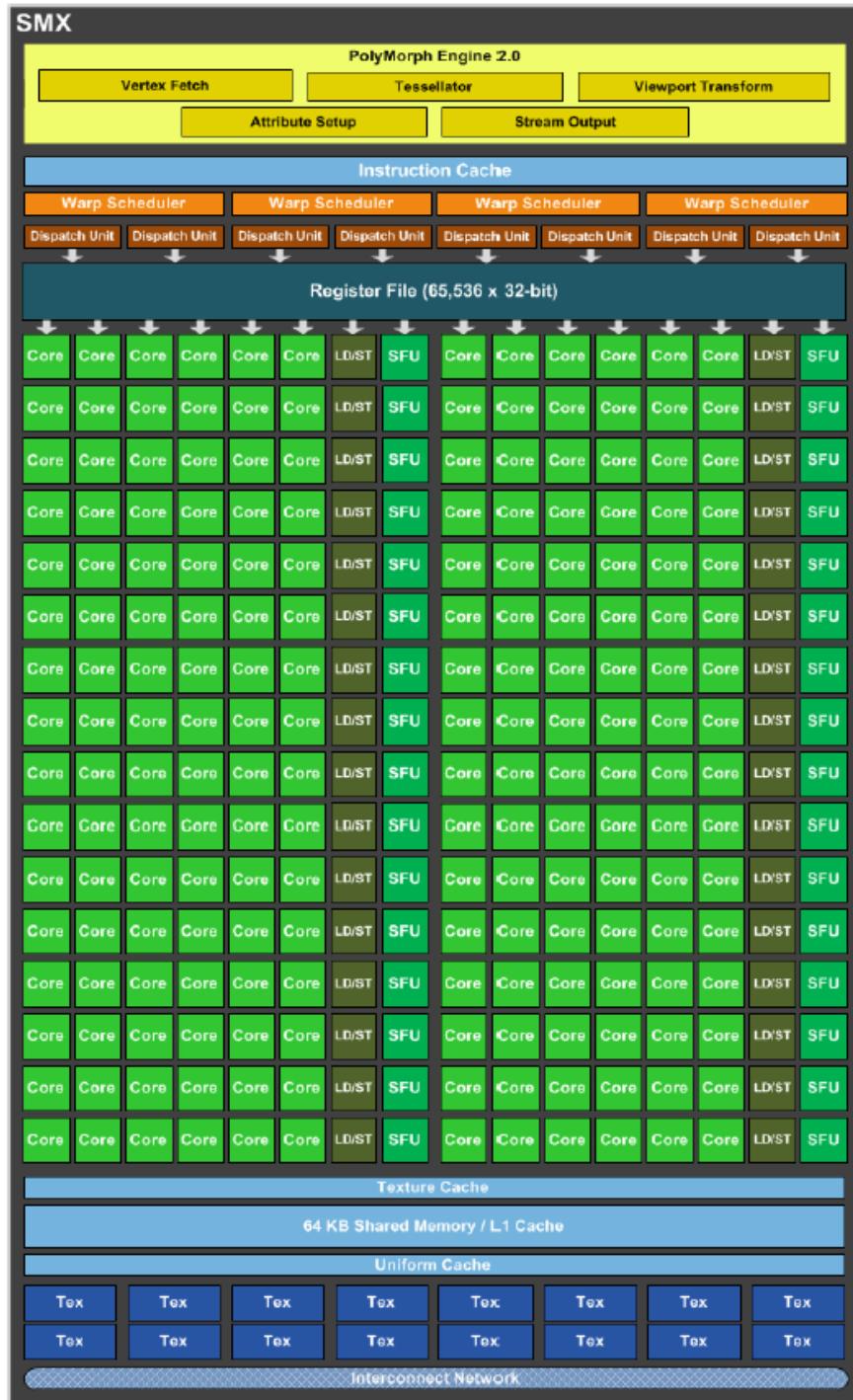


## Multiprocessor: SMX (CC 3.0)

- 192 CUDA cores  
( $192 = 6 * 32$ )
- 32 LD/ST units
- 32 SFUs
- 16 texture units

Two dispatch units per warp scheduler exploit ILP  
(*instruction-level parallelism*)

*Can dual-issue ALU instructions!  
("superscalar")*



# GK110 SMX

## Multiprocessor: SMX (CC 3.5)

- 192 CUDA cores  
 $(192 = 6 * 32)$
  - 64 DP units
  - 32 LD/ST units
  - 32 SFUs
  - 16 texture units

## New read-only data cache (48KB)





## K.3.1. Architecture

An SM has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory.

There is an L1 cache for each SM and an L2 cache shared by all SMs. The L1 cache is used to cache accesses to local memory, including temporary register spills. The L2 cache is used to cache accesses to local and global memory. The cache behavior (e.g., whether reads are cached in both L1 and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load or store instruction. Some devices of compute capability 3.5 and devices of compute capability 3.7 allow opt-in to caching of global memory in both L1 and L2 via compiler options.

The same on-chip memory is used for both L1 and shared memory: It can be configured as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory and 48 KB of L1 cache or as 32 KB of shared memory and 32 KB of L1 cache, using `cudaFuncSetCacheConfig()`/`cuFuncSetCacheConfig()`:

# Compute Capab. 3.x (Kepler, Part 2)



**Note:** Devices of compute capability 3.7 add an additional 64 KB of shared memory to each of the above configurations, yielding 112 KB, 96 KB, and 80 KB shared memory per SM, respectively. However, the maximum shared memory per thread block remains 48 KB.

Applications may query the L2 cache size by checking the `l2CacheSize` device property (see [Device Enumeration](#)). The maximum L2 cache size is 1.5 MB.

Each SM has a read-only data cache of 48 KB to speed up reads from device memory. It accesses this cache either directly (for devices of compute capability 3.5 or 3.7), or via a texture unit that implements the various addressing modes and data filtering mentioned in [Texture and Surface Memory](#). When accessed via the texture unit, the read-only data cache is also referred to as texture cache.



## K.3.2. Global Memory

Global memory accesses for devices of compute capability 3.x are cached in L2 and for devices of compute capability 3.5 or 3.7, may also be cached in the read-only data cache described in the previous section; they are normally not cached in L1. Some devices of compute capability 3.5 and devices of compute capability 3.7 allow opt-in to caching of global memory accesses in L1 via the `-Xptxas -dlcm=ca` option to nvcc.

A cache line is 128 bytes and maps to a 128 byte aligned segment in device memory. Memory accesses that are cached in both L1 and L2 are serviced with 128-byte memory transactions, whereas memory accesses that are cached in L2 only are serviced with 32-byte memory transactions. Caching in L2 only can therefore reduce over-fetch, for example, in the case of scattered memory accesses.

If the size of the words accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently:

- ▶ Two memory requests, one for each half-warp, if the size is 8 bytes,
- ▶ Four memory requests, one for each quarter-warp, if the size is 16 bytes.

# Compute Capab. 3.x (Kepler, Part 4)



Each memory request is then broken down into cache line requests that are issued independently. A cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.

Note that threads can access any words in any order, including the same words.

If a non-atomic instruction executed by a warp writes to the same location in global memory for more than one of the threads of the warp, only one thread performs a write and which thread does it is undefined.

Data that is read-only for the entire lifetime of the kernel can also be cached in the read-only data cache described in the previous section by reading it using the `__ldg()` function (see [Read-Only Data Cache Load Function](#)). When the compiler detects that the read-only condition is satisfied for some data, it will use `__ldg()` to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the `const` and `__restrict__` qualifiers increases the likelihood that the compiler will detect the read-only condition.

[Figure 21](#) shows some examples of global memory accesses and corresponding memory transactions.

# Maxwell (GM) Architecture

## Multiprocessor: SMM (CC 5.x)

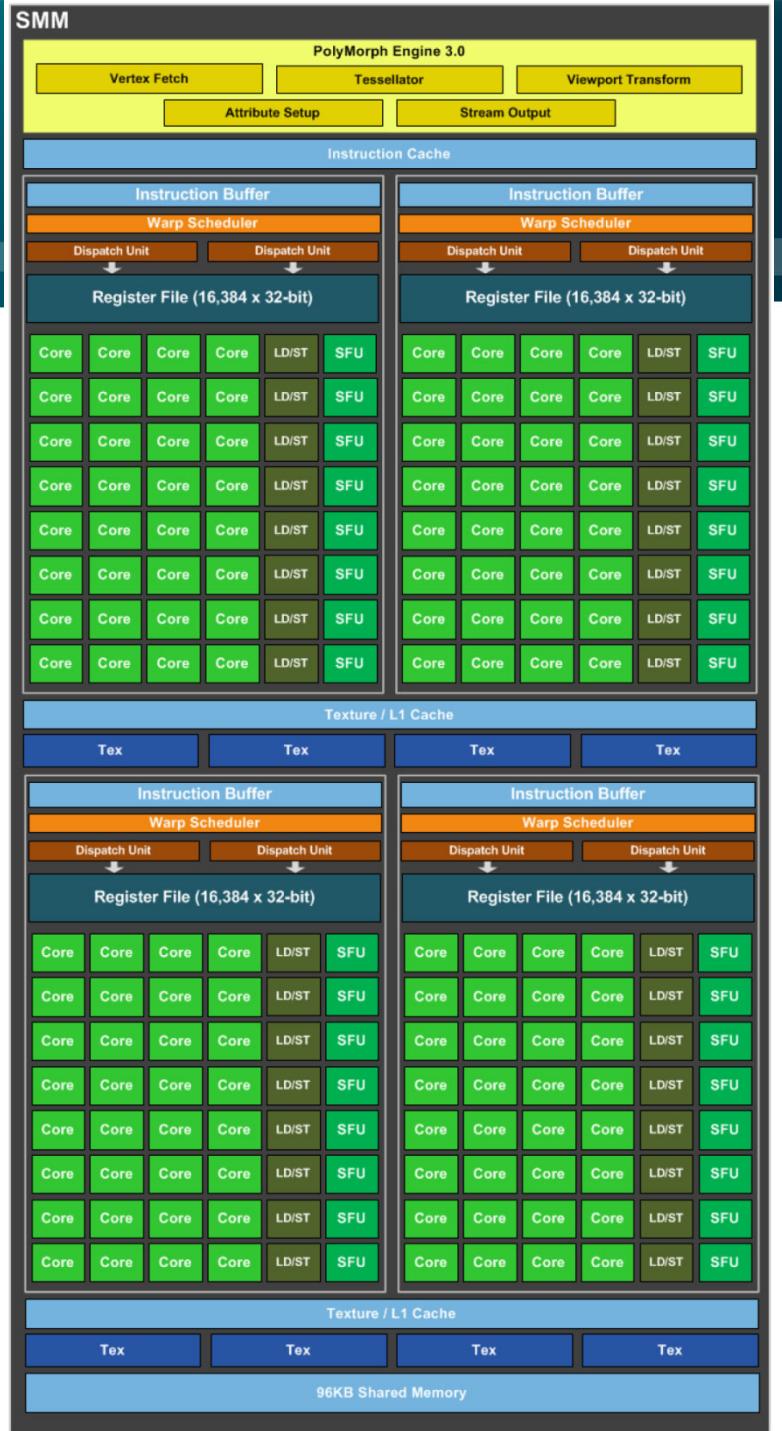
- 128 CUDA cores
- 4 DP units; 32 LD/ST units; 32 SFUs
- 8 texture units

## 4 partitions inside SMM

- 32 CUDA cores each
- 8 LD/ST units; 8 SFUs each
- Each has its own register file, warp scheduler, two dispatch units (*but cannot dual-issue ALU insts.!*)

Shared memory and L1 cache now separate!

- L1 cache shares with texture cache
- Shared memory is its own space





## K.4.1. Architecture

An SM has:

- ▶ a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- ▶ a unified L1/texture cache of 24 KB used to cache reads from global memory,
- ▶ 64 KB of shared memory for devices of compute capability 5.0 or 96 KB of shared memory for devices of compute capability 5.2.

The unified L1/texture cache is also used by the texture unit that implements the various addressing modes and data filtering mentioned in [Texture and Surface Memory](#).

There is also an L2 cache shared by all SMs that is used to cache accesses to local or global memory, including temporary register spills. Applications may query the L2 cache size by checking the `l2CacheSize` device property (see [Device Enumeration](#)).

The cache behavior (e.g., whether reads are cached in both the unified L1/texture cache and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load instruction.



## K.4.2. Global Memory

Global memory accesses are always cached in L2 and caching in L2 behaves in the same way as for devices of compute capability 3.x (see [Global Memory](#)).

Data that is read-only for the entire lifetime of the kernel can also be cached in the unified L1/texture cache described in the previous section by reading it using the `_ldg()` function (see [Read-Only Data Cache Load Function](#)). When the compiler detects that the read-only condition is satisfied for some data, it will use `_ldg()` to read it. The compiler might not always be able to detect that the read-only condition is satisfied for some data. Marking pointers used for loading such data with both the `const` and `_restrict_` qualifiers increases the likelihood that the compiler will detect the read-only condition.

Data that is not read-only for the entire lifetime of the kernel cannot be cached in the unified L1/texture cache for devices of compute capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms:

# Compute Capab. 5.x (Maxwell, Part 3)



Data that is not read-only for the entire lifetime of the kernel cannot be cached in the unified L1/texture cache for devices of compute capability 5.0. For devices of compute capability 5.2, it is, by default, not cached in the unified L1/texture cache, but caching may be enabled using the following mechanisms:

- ▶ Perform the read using inline assembly with the appropriate modifier as described in the PTX reference manual;
- ▶ Compile with the `-Xptxas -dlcm=ca` compilation flag, in which case all reads are cached, except reads that are performed using inline assembly with a modifier that disables caching;
- ▶ Compile with the `-Xptxas -fscm=ca` compilation flag, in which case all reads are cached, including reads that are performed using inline assembly regardless of the modifier used.

When caching is enabled using one of the three mechanisms listed above, devices of compute capability 5.2 will cache global memory reads in the unified L1/texture cache for all kernel launches except for the kernel launches for which thread blocks consume too much of the SM's register file. These exceptions are reported by the profiler.



## K.4.3. Shared Memory

Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per clock cycle.

A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word (even though the two addresses fall in the same bank). In that case, for read accesses, the word is broadcast to the requesting threads and for write accesses, each address is written by only one of the threads (which thread performs the write is undefined).

[Figure 22](#) shows some examples of strided access.

[Figure 23](#) shows some examples of memory read accesses that involve the broadcast mechanism.

# NVIDIA Pascal GP100 SM



Multiprocessor: SM (CC 6.0)

- 64 CUDA cores
- 32 DP units
- 16 LD/ST units
- 16 SFUs
- 4 texture units



2 partitions inside SM

- 32 CUDA cores each; 16 DP units each; 8 LD/ST units each; 8 SFUs each
- Each has its own register file, warp scheduler, two dispatch units  
*(but cannot dual-issue ALU (single precision core) insts.!)*

# NVIDIA Pascal GP104 SM

## Multiprocessor: SM (CC 6.1/6.2)

- 128 CUDA cores
- 32 LD/ST units
- 32 SFUs
- 8 texture units

## 4 partitions inside SM

- 32 CUDA cores; 8 LD/ST units; 8 SFUs
- Each has its own register file, warp scheduler, two dispatch units  
*(but cannot dual-issue ALU insts.!)*





## K.5.1. Architecture

An SM has:

- ▶ a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- ▶ a unified L1/texture cache for reads from global memory of size 24 KB (6.0 and 6.2) or 48 KB (6.1),
- ▶ a shared memory of size 64 KB (6.0 and 6.2) or 96 KB (6.1).

The unified L1/texture cache is also used by the texture unit that implements the various addressing modes and data filtering mentioned in [Texture and Surface Memory](#).

There is also an L2 cache shared by all SMs that is used to cache accesses to local or global memory, including temporary register spills. Applications may query the L2 cache size by checking the `l2CacheSize` device property (see [Device Enumeration](#)).

The cache behavior (e.g., whether reads are cached in both the unified L1/texture cache and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load instruction.



## K.5.2. Global Memory

Global memory behaves the same way as in devices of compute capability 5.x (See [Global Memory](#)).

## K.5.3. Shared Memory

Shared memory behaves the same way as in devices of compute capability 5.x (See [Shared Memory](#)).

# NVIDIA Volta SM

## Multiprocessor: SM (CC 7.0)

- 64 FP32 + 64 INT32 cores
- 32 FP64 cores
- 32 LD/ST units; 16 SFUs
- 8 tensor cores  
(FP16/FP32 mixed-precision)

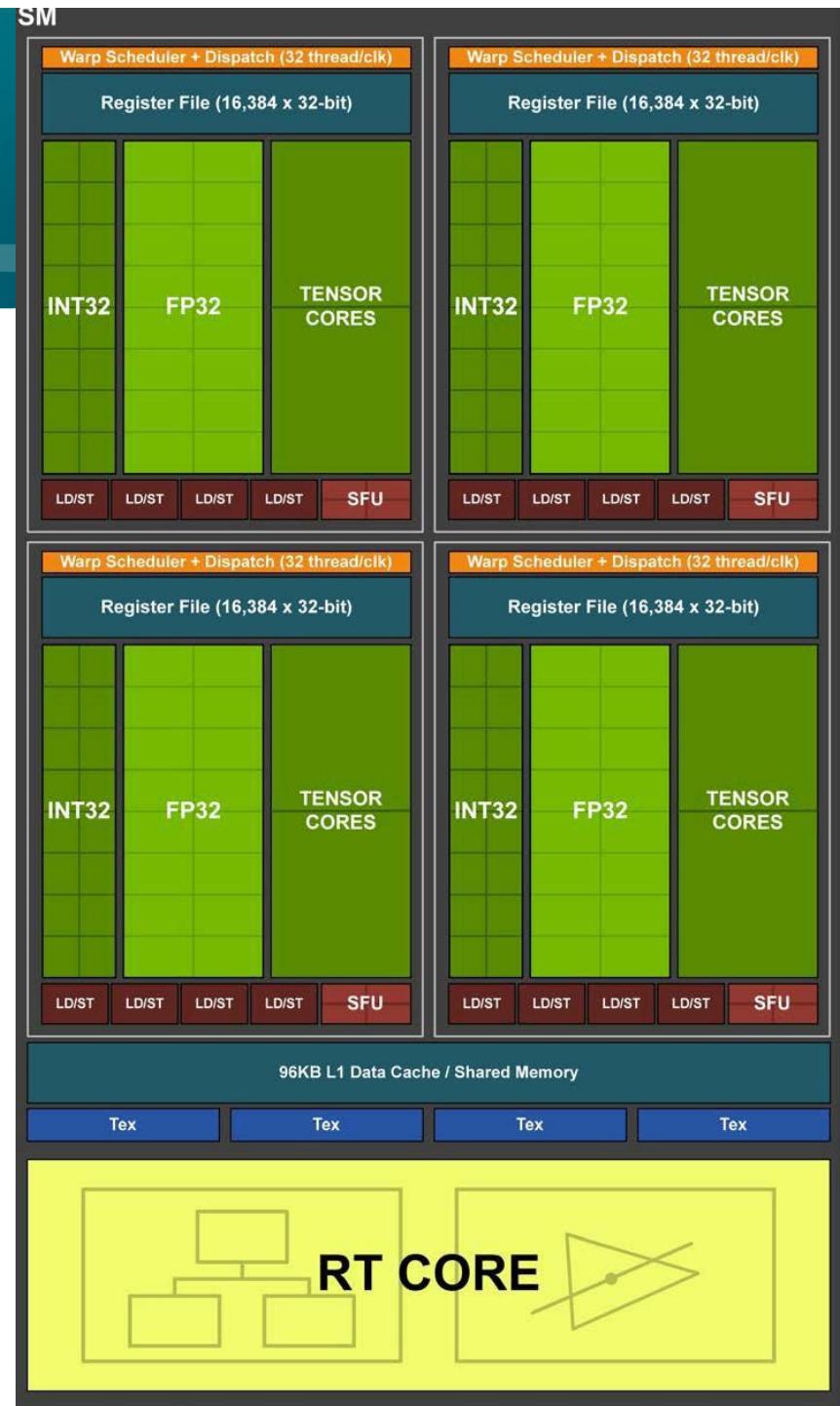
## 4 partitions inside SM

- 16 FP32 + 16 INT32 cores each
- 8 FP64 cores each
- 8 LD/ST units; 4 SFUs each
- 2 tensor cores each
- Each has: warp scheduler, dispatch unit, register file



# NVIDIA Turing SM

- Multiprocessor: SM (CC 7.5)
- 64 FP32 + INT32 cores
  - 2 (!) FP64 cores
  - 8 Turing tensor cores  
(FP16/32, INT4/8 mixed-precision)
  - 1 RT (ray tracing) core
- 4 partitions inside SM
- 16 FP32 + INT32 cores each
  - 4 LD/ST units; 4 SFUs each
  - 2 Turing tensor cores each
  - Each has: warp scheduler, dispatch unit, 16K register file





## K.6.1. Architecture

An SM has:

- ▶ a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- ▶ a unified data cache and shared memory with a total size of 128 KB (Volta) or 96 KB (Turing).

Shared memory is partitioned out of unified data cache, and can be configured to various sizes (See [Shared Memory](#).) The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in [Texture and Surface Memory](#).



## K.6.3. Global Memory

Global memory behaves the same way as in devices of compute capability 5.x (See [Global Memory](#)).

## K.6.4. Shared Memory

Similar to the [Kepler architecture](#), the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *Volta* architecture (compute capability 7.0), the unified data cache has a size of 128 KB, and the shared memory capacity can be set to 0, 8, 16, 32, 64 or 96 KB. For the *Turing* architecture (compute capability 7.5), the unified data cache has a size of 96 KB, and the shared memory capacity can be set to either 32 KB or 64 KB. Unlike *Kepler*, the driver automatically configures the shared memory capacity for each kernel to avoid shared memory occupancy bottlenecks while also allowing concurrent execution with already launched kernels where possible. In most cases, the driver's default behavior should provide optimal performance.

# Compute Capab. 7.x (Volta/Turing, Part 3)



Because the driver is not always aware of the full workload, it is sometimes useful for applications to provide additional hints regarding the desired shared memory configuration. For example, a kernel with little or no shared memory use may request a larger carveout in order to encourage concurrent execution with later kernels that require more shared memory. The new `cudaFuncSetAttribute()` API allows applications to set a preferred shared memory capacity, or `carveout`, as a percentage of the maximum supported shared memory capacity (96 KB for *Volta*, and 64 KB for *Turing*).

`cudaFuncSetAttribute()` relaxes enforcement of the preferred shared capacity compared to the legacy `cudaFuncSetCacheConfig()` API introduced with [Kepler](#). The legacy API treated shared memory capacities as hard requirements for kernel launch. As a result, interleaving kernels with different shared memory configurations would needlessly serialize launches behind shared memory reconfigurations. With the new API, the `carveout` is treated as a hint. The driver may choose a different configuration if required to execute the function or to avoid thrashing.

# Compute Capab. 7.x (Volta/Turing, Part 4)



```
// Device code
__global__ void MyKernel(...)
{
    __shared__ float buffer[BLOCK_DIM];
    ...
}

// Host code
int carveout = 50; // prefer shared memory capacity 50% of maximum
// Named Carveout Values:
// carveout = cudaSharedmemCarveoutDefault;    // (-1)
// carveout = cudaSharedmemCarveoutMaxL1;        // (0)
// carveout = cudaSharedmemCarveoutMaxShared; // (100)
cudaFuncSetAttribute(MyKernel, cudaFuncAttributePreferredSharedMemoryCarveout,
                     carveout);
MyKernel <<<gridDim, BLOCK_DIM>>>(...);
```

In addition to an integer percentage, several convenience enums are provided as listed in the code comments above. Where a chosen integer percentage does not map exactly to a supported capacity (SM 7.0 devices support shared capacities of 0, 8, 16, 32, 64, or 96 KB), the next larger capacity is used. For instance, in the example above, 50% of the 96 KB maximum is 48 KB, which is not a supported shared memory capacity. Thus, the preference is rounded up to 64 KB.

# Compute Capab. 7.x (Volta/Turing, Part 5)



Compute capability 7.x devices allow a single thread block to address the full capacity of shared memory: 96 KB on *Volta*, 64 KB on *Turing*. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, as such they must use dynamic shared memory (rather than statically sized arrays) and require an explicit opt-in using `cudaFuncSetAttribute()` as follows.

```
// Device code
__global__ void MyKernel(...)
{
    extern __shared__ float buffer[];
    ...
}

// Host code
int maxbytes = 98304; // 96 KB
cudaFuncSetAttribute(MyKernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
    maxbytes);
MyKernel <<<gridDim, blockDim, maxbytes>>>(...);
```

Otherwise, shared memory behaves the same way as for devices of compute capability 5.x (See [Shared Memory](#)).

# NVIDIA GA100 SM

Multiprocessor: SM (CC 8.0)

- 64 FP32 + 64 INT32 cores
- 32 FP64 cores
- 4 3<sup>rd</sup> gen tensor cores
- 1 2<sup>nd</sup> gen RT (ray tracing) core

4 partitions inside SM

- 16 FP32 + 16 INT32 cores
- 8 FP64 cores
- 8 LD/ST units; 4 SFUs each
- 1 3<sup>rd</sup> gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file



# NVIDIA GA10x SM

Multiprocessor: SM (CC 8.6)

- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores
- 4 3<sup>rd</sup> gen tensor cores
- 1 2<sup>nd</sup> gen RT (ray tracing) core

4 partitions inside SM

- 32 (16+16) FP32 + 16 INT32 cores
- 4 LD/ST units; 4 SFUs each
- 1 3<sup>rd</sup> gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file



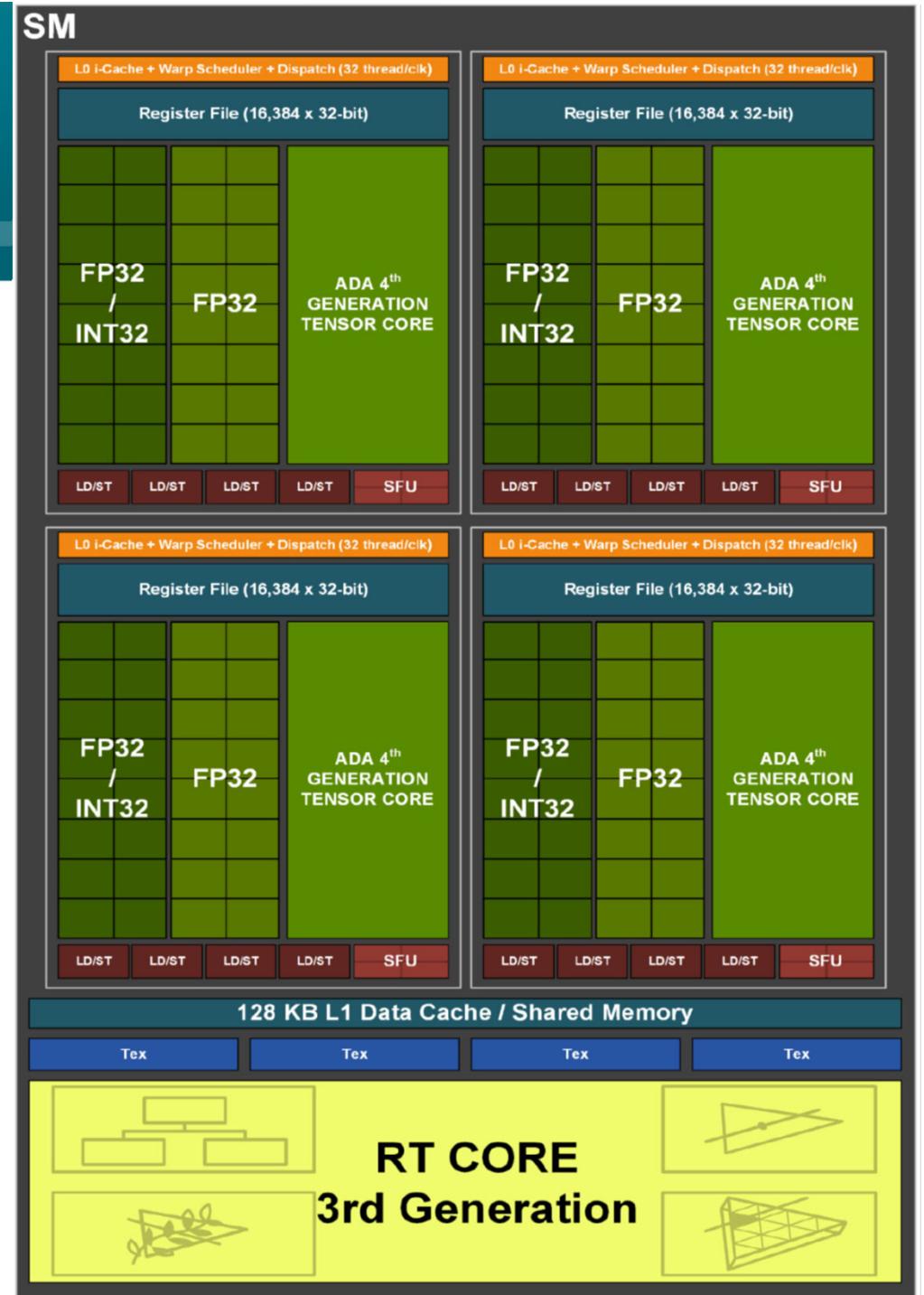
# NVIDIA AD102 SM

Multiprocessor: SM (CC 8.9)

- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores (not in diagram)
- 4x 4<sup>th</sup> gen tensor cores
- 1x 3<sup>rd</sup> gen RT (ray tracing) core
- ++ thread block clusters, FP8, ... (?)

4 partitions inside SM

- 32 (16+16) FP32 + 16 INT32 cores
- 4x LD/ST units; 4 SFUs each
- 1x 4<sup>th</sup> gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file





## K.7.1. Architecture

An SM has:

- ▶ a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- ▶ a unified data cache and shared memory with a total size of 192 KB for devices of compute capability 8.0 and 8.7 (1.5x Volta's 128 KB capacity) and 128 KB for devices of compute capabilities 8.6 and 8.9.

Shared memory is partitioned out of the unified data cache, and can be configured to various sizes (see [Shared Memory](#) section). The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in [Texture and Surface Memory](#).



## K.7.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See [Global Memory](#)).

## K.7.3. Shared Memory

Similar to the [Volta architecture](#), the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *NVIDIA Ampere GPU architecture*, the unified data cache has a size of 192 KB for devices of compute capability 8.0 and 128 KB for devices of compute capability 8.6 and 8.9. The shared memory capacity can be set to 0, 8, 16, 32, 64, 100, 132 or 164 KB for devices of compute capability 8.0, and to 0, 8, 16, 32, 64 or 100 KB for devices of compute capabilities 8.6 and 8.9.

An application can set the `carveout`, i.e., the preferred shared memory capacity, with the `cudaFuncSetAttribute()`.

```
cudaFuncSetAttribute(kernel_name, cudaFuncAttributePreferredSharedMemoryCarveout,  
carveout);
```

# Compute Capab. 8.x (Ampere/Ada, Part 3)



The API can specify the carveout either as an integer percentage of the maximum supported shared memory capacity of 164 KB for devices of compute capability 8.0 and 100 KB for devices of compute capabilities 8.6 and 8.9 respectively, or as one of the following values: `{cudaSharedmemCarveoutDefault, cudaSharedmemCarveoutMaxL1, or cudaSharedmemCarveoutMaxShared}`. When using a percentage, the carveout is rounded up to the nearest supported shared memory capacity. For example, for devices of compute capability 8.0, 50% will map to a 100 KB carveout instead of an 82 KB one. Setting the `cudaFuncAttributePreferredSharedMemoryCarveout` is considered a hint by the driver; the driver may choose a different configuration, if needed.

Devices of compute capability 8.0 allow a single thread block to address up to 163 KB of shared memory, while devices of compute capabilities 8.6 and 8.9 allow up to 99 KB of shared memory. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, and must use dynamic shared memory rather than statically sized shared memory arrays. These kernels require an explicit opt-in by using `cudaFuncSetAttribute()` to set the `cudaFuncAttributeMaxDynamicSharedMemorySize`; see [Shared Memory](#) for the Volta architecture.

Note that the maximum amount of shared memory per thread block is smaller than the maximum shared memory partition available per SM. The 1 KB of shared memory not made available to a thread block is reserved for system use.

# NVIDIA GH100 SM

## Multiprocessor: SM (CC 9.0)

- 128 FP32 + 64 INT32 cores
- 64 FP64 cores
- 4x 4<sup>th</sup> gen tensor cores
- ++ thread block clusters, DPX insts., FP8, TMA

## 4 partitions inside SM

- 32 FP32 + 16 INT32 cores
- 16 FP64 cores
- 8x LD/ST units; 4 SFUs each
- 1x 4<sup>th</sup> gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file





## K.8.1. Architecture

An SM has:

- ▶ a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- ▶ a unified data cache and shared memory with a total size of 256 KB for devices of compute capability 9.0 (1.33x NVIDIA Ampere GPU Architecture's 192 KB capacity).

Shared memory is partitioned out of the unified data cache, and can be configured to various sizes (see [Shared Memory](#) section). The remaining data cache serves as an L1 cache and is also used by the texture unit that implements the various addressing and data filtering modes mentioned in [Texture and Surface Memory](#).

## K.8.2. Global Memory

Global memory behaves the same way as for devices of compute capability 5.x (See [Global Memory](#)).



## K.8.3. Shared Memory

Similar to the [NVIDIA Ampere GPU architecture](#), the amount of the unified data cache reserved for shared memory is configurable on a per kernel basis. For the *NVIDIA H100 Tensor Core GPU architecture*, the unified data cache has a size of 256 KB for devices of compute capability 9.0. The shared memory capacity can be set to 0, 8, 16, 32, 64, 100, 132, 164, 196 or 228 KB.

As with the [NVIDIA Ampere GPU architecture](#), an application can configure its preferred shared memory capacity, i.e., the `carveout`. Devices of compute capability 9.0 allow a single thread block to address up to 227 KB of shared memory. Kernels relying on shared memory allocations over 48 KB per block are architecture-specific, and must use dynamic shared memory rather than statically sized shared memory arrays. These kernels require an explicit opt-in by using `cudaFuncSetAttribute()` to set the `cudaFuncAttributeMaxDynamicSharedMemorySize`; see [Shared Memory](#) for the Volta architecture.

Note that the maximum amount of shared memory per thread block is smaller than the maximum shared memory partition available per SM. The 1 KB of shared memory not made available to a thread block is reserved for system use.

Thank you.