In this second part of the tutorial, having explored the basic CUDA programming model in part one, we will now take a closer look at how the programming model maps to the hardware architecture and discuss various low-level aspects that are crucial for performance.

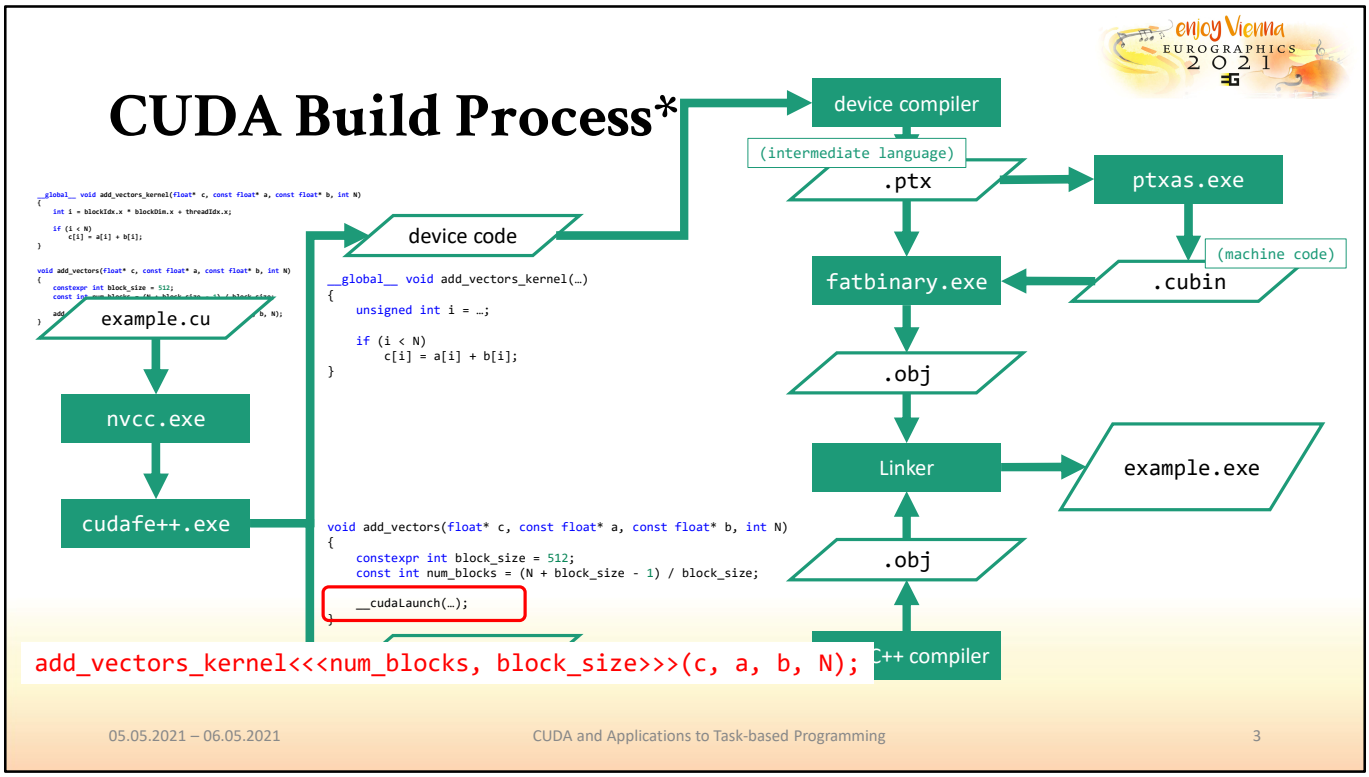
example.cu

```
__global__ void add_vectors_kernel(float* c, const float* a, const float* b, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N)
        c[i] = a[i] + b[i];
}
```

GPU code

```
void add_vectors(float* c, const float* a, const float* b, int N)
{
    constexpr int block_size = 512;
    const int num_blocks = (N + block_size - 1) / block_size;

    add_vectors_kernel<<<num_blocks, block_size>>>(c, a, b, N);
}
```

CPU code



compiler magic

Parallel Thread Execution (PTX)

- virtual GPU architecture
- intermediate language target

- translated into actual machine code
  - by `ptxas.exe`
  - by driver at runtime (JIT)

- analogous to LLVM, SPIR-V, DXBC, DXIL
- well-documented

```
.visible .entry _Z10add_kernelPfPKfS1_j(
        .param .u64 _Z10add_kernelPfPKfS1_j_param_0,
        .param .u64 _Z10add_kernelPfPKfS1_j_param_1,
        .param .u64 _Z10add_kernelPfPKfS1_j_param_2,
        .param .u32 _Z10add_kernelPfPKfS1_j_param_3
)
{
Lfunc_begin0:

        ld.param.u64    %rd1, [_Z10add_kernelPfPKfS1_j_param_
        ld.param.u64    %rd2, [_Z10add_kernelPfPKfS1_j_param_
        ld.param.u64    %rd3, [_Z10add_kernelPfPKfS1_j_param_
        ld.param.u32    %r2, [_Z10add_kernelPfPKfS1_j_param_3
Ltmp0:
        mov.u32         %r3, %ctaid.x;
        mov.u32         %r4, %ntid.x;
        mov.u32         %r5, %tid.x;
        mad.lo.s32      %r1, %r3, %r4, %r5;
        setp.ge.u32     %p1, %r1, %r2;
        @%p1 bra        LBB0_2;

        cvta.to.global.u64      %rd4, %rd2;
Ltmp1:
        mul.wide.u32    %rd5, %r1, 4;
        add.s64         %rd6, %rd4, %rd5;
Ltmp2:
        cvta.to.global.u64      %rd7, %rd3;
Ltmp3:
        add.s64         %rd8, %rd7, %rd5;
        ld.global.f32   %f1, [%rd8];
        ld.global.f32   %f2, [%rd6];
        add.f32         %f3, %f2, %f1;
Ltmp4:
        cvta.to.global.u64      %rd9, %rd1;
Ltmp5:
        add.s64         %rd10, %rd9, %rd5;
        st.global.f32   [%rd10], %f3;
```

SASS (Shader Assembly?)

- actual machine instructions
- obtained by disassembling `.cubin`
- not really documented
  - superficial documentation in CUDA Toolkit
  - `nvdisasm.exe`
  - some insights to be found in NVIDIA patents
  - various reverse engineering efforts

```
_Z10add_kernelPfPKfS1_j:
 MOV R1, c[0x0][0x20]
 S2R R0, SR_CTAID.X
 S2R R2, SR_TID.X
 XMAD.MRG R3, R0.reuse, c[0x0] [0x8].H1, RZ
 XMAD R2, R0.reuse, c[0x0] [0x8], R2
 XMAD.PSL.CBCC R0, R0.H1, R3.H1, R2
 ISETP.GE.U32.AND P0, PT, R0, c[0x0][0x158], PT
 NOP
 @P0 EXIT
 SHL R6, R0.reuse, 0x2
 SHR.U32 R0, R0, 0x1e
 IADD R4.CC, R6.reuse, c[0x0][0x148]
 IADD.X R5, R0.reuse, c[0x0][0x14c]
 IADD R2.CC, R6, c[0x0][0x150]
 LDG.E R4, [R4]
 IADD.X R3, R0, c[0x0][0x154]
 LDG.E R2, [R2]
 IADD R6.CC, R6, c[0x0][0x140]
 IADD.X R7, R0, c[0x0][0x144]
 FADD R0, R2, R4
 STG.E [R6], R0
 NOP
 EXIT
.L_1:
 BRA `(.L_1)
.L_26:
```

CUDA Build Process: Summary

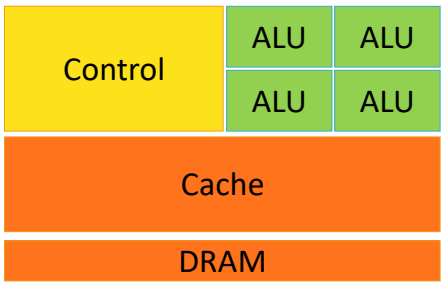
- factor CUDA C++ into host and device parts
  - compiled separately
- generated host code
  - takes care of loading matching GPU binary stored in `.exe`
  - translate `kernel<<<…>>>(…)` syntax into API calls
- "Fat Binary" can contain both
  - PTX for various compute capabilities
    - allows the binary to target unknown architecture
  - precompiled machine code for specific GPU architectures
    - optimal performance on certain known devices

CUDA: Level 2

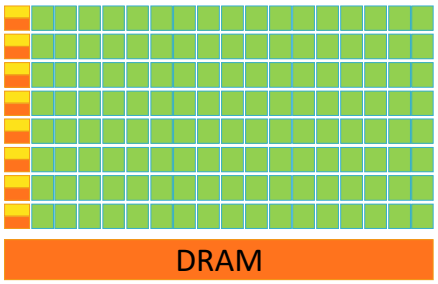
GPU Architecture Recap

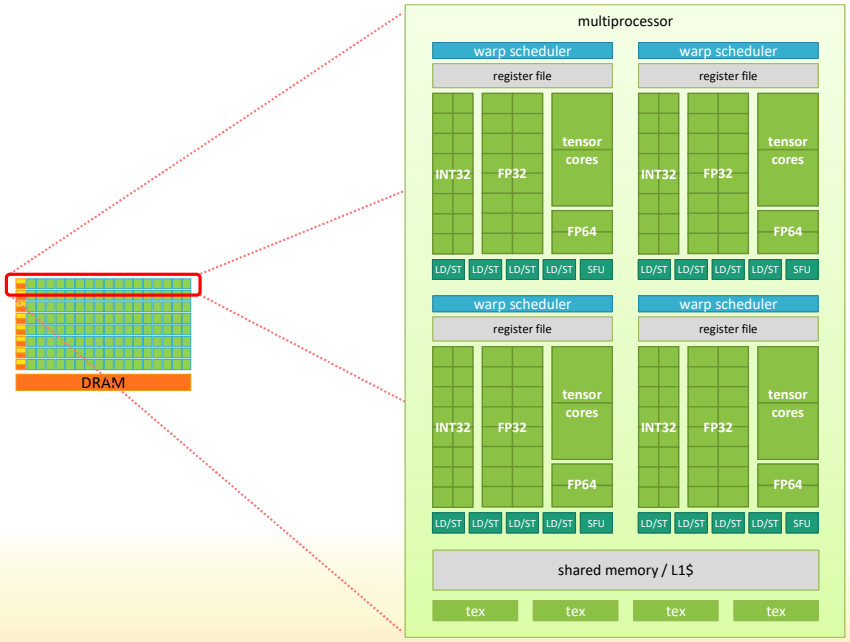
- Basic Premise: We have more work than we have cores.
- instead of waiting for long-running operation: switch to other task
- Prerequisites:
  - enough work → massively parallel workload
  - very fast context switching

CPU: minimize latency



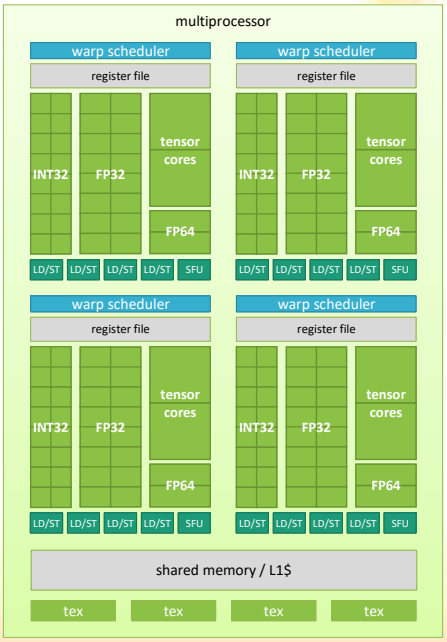
GPU: maximize throughput



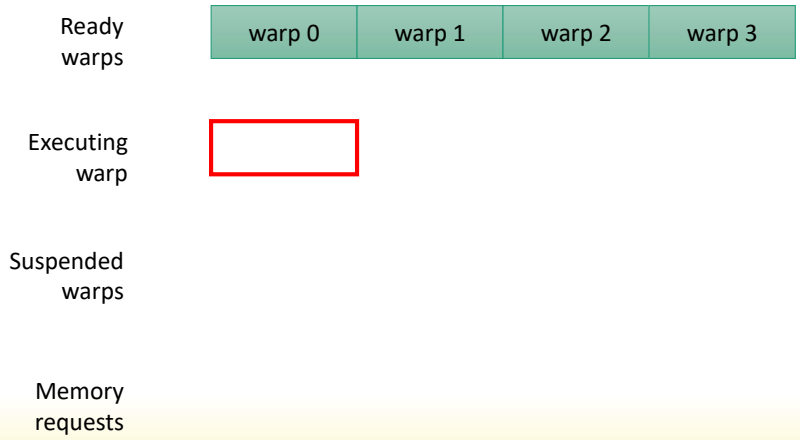

DRAM
multiprocessor
warp scheduler
register file
INT32
FP32
tensor cores
FP64
LD/ST
SFU
shared memory / L1$
tex

Warp Scheduling

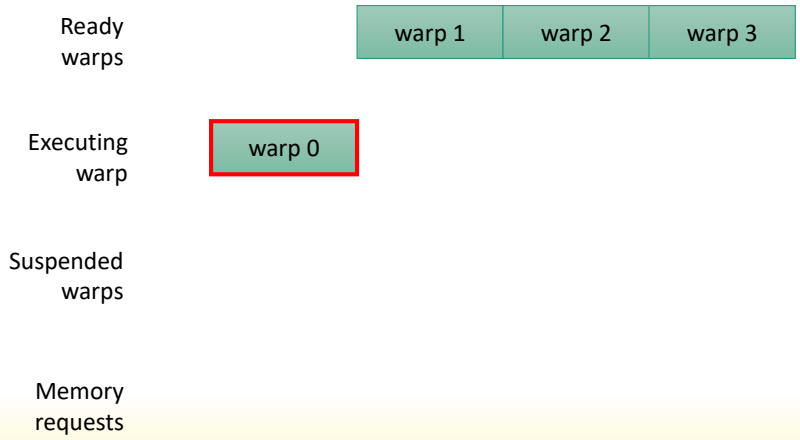
- each warp scheduler can schedule to a number of
  - ALUs
    - FP32, INT32, FP64, tensor cores
  - special function units
    - sqrt, exp, …
  - load/store units
- thread context for multiple warps resident on chip
  - latency hiding
- every clock cycle:
  - warp scheduler elects eligible warp
  - schedules instruction from that warp



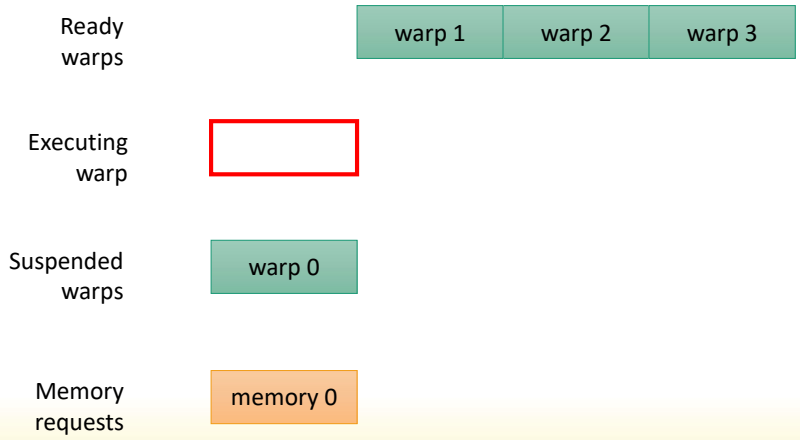
Warp Scheduling: Latency Hiding



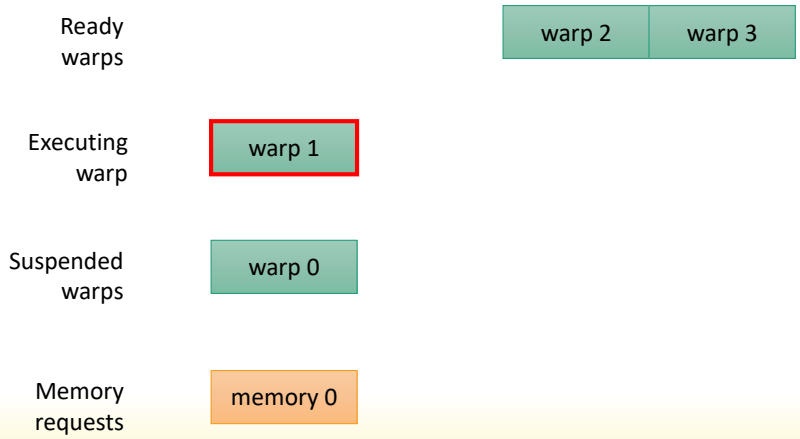
Warp Scheduling: Latency Hiding



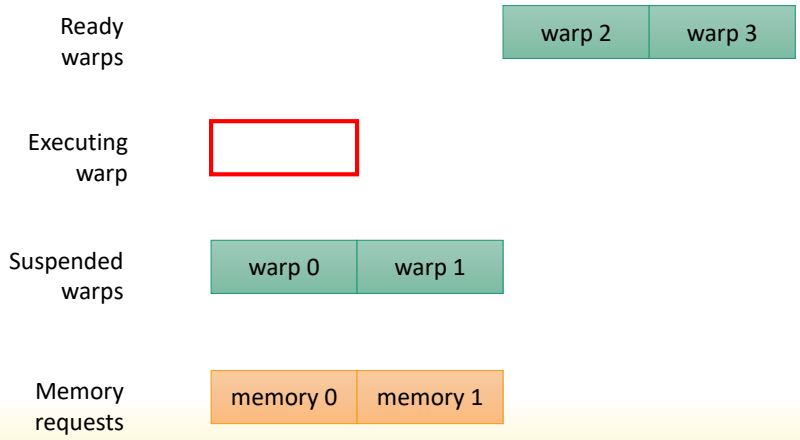
Warp Scheduling: Latency Hiding



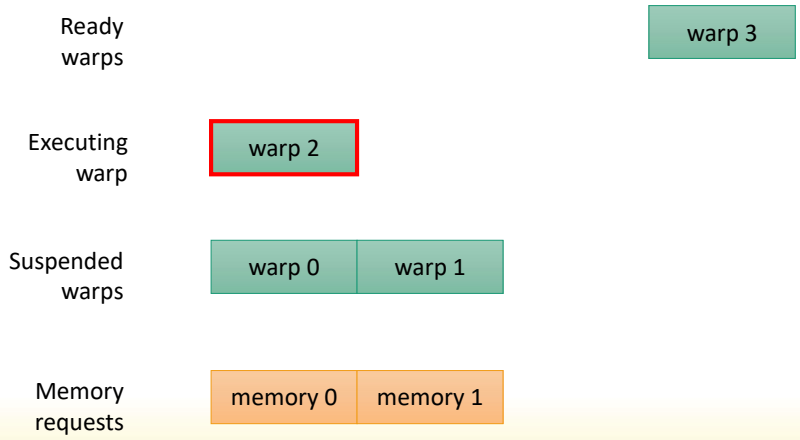
Warp Scheduling: Latency Hiding



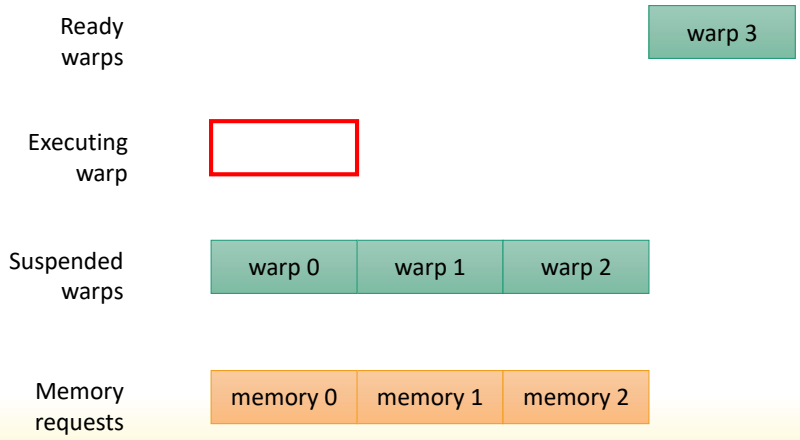
Warp Scheduling: Latency Hiding



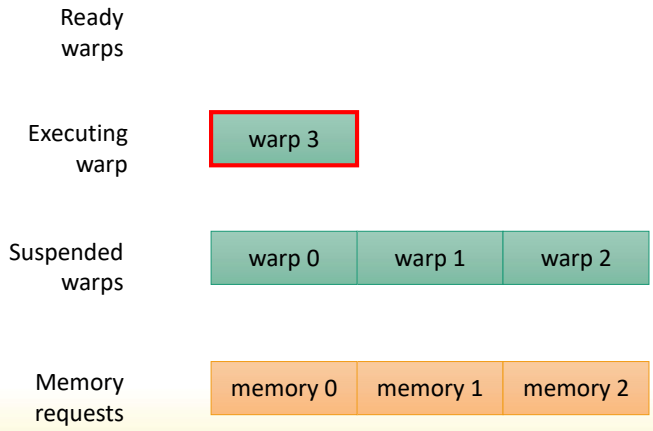
Warp Scheduling: Latency Hiding



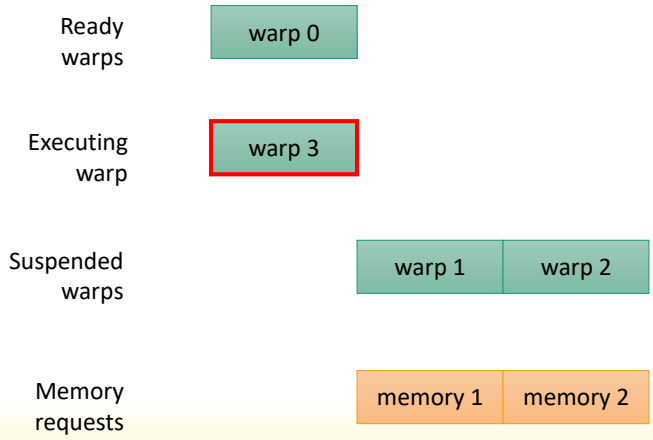
Warp Scheduling: Latency Hiding



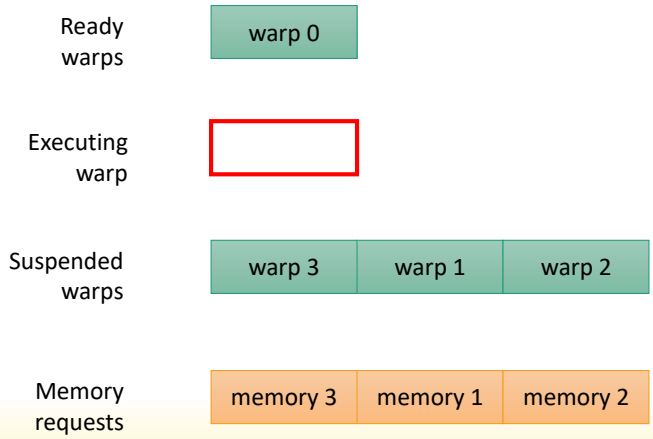
Warp Scheduling: Latency Hiding



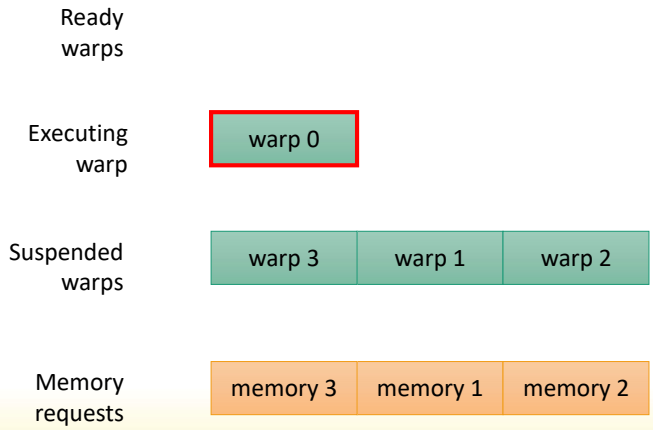
Warp Scheduling: Latency Hiding



Warp Scheduling: Latency Hiding

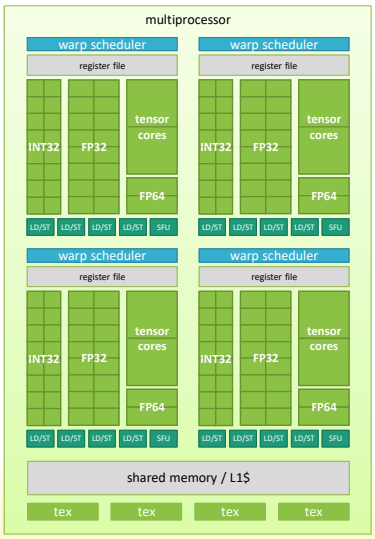


Warp Scheduling: Latency Hiding



Warp Scheduling: Control Flow

- SIMT/SIMD execution
  - one control flow, multiple data paths
  - spend more silicon on raw computation
- What about branches?



Control Flow

- unconditional branches
  - all threads jump to target
  - no problem

```
f();

goto x;
```

- uniform branches
  - all threads conditionally jump to target
  - no problem

```
if (blockIdx.x < 16)
{
    …
}
```

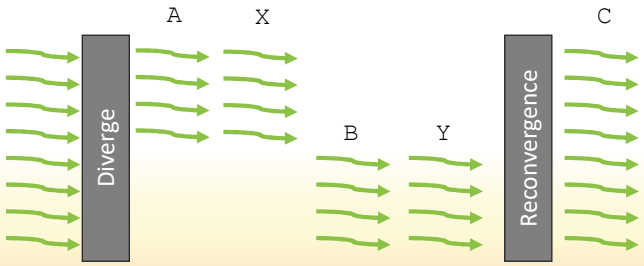
- non-uniform branches
  - some threads jump to target, others don't
  - problem

```
if (threadIdx.x < 16)
{
    …
}
```

Classic SIMT Execution

- 32 threads (1 warp) execute in **SIMT** fashion
  - 1 program counter shared per warp
  - divergent paths leave threads inactive
    - whole warp has to execute each branch sequentially
  - reconverge after divergent section

```
if(threadIdx.x & 0x4)
{
    A();
    X();
}
else
{
    B();
    Y();
}
C();
```



25

Control Flow

- implement via Predication
  - mask off threads that are not in active branch
- what about branches in branches?
  - what about branches in branches in branches?
    - …
- Branch Stack
  - push mask of active threads
  - run first half
  - pop mask of active threads
  - run other half

```
if (A)
{
    if (B)
    {
        if (C)
        {
            …
        }
        else
        {
            …
        }
    }
    else
    {
        …
    }
}
```

Example 1

```
__global__ void test(int* out, int N)
{
    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < N)
    {
        out[tid] = N;
    }
}
```

https://godbolt.org/z/7csfx5

Example 1: Predication

```
test(int*, int):
 MOV R1, c[0x0][0x44]
 S2R R0, SR_CTAID.X
 S2R R3, SR_TID.X
 IMAD R0, R0, c[0x0][0x28], R3
 ISETP.GE.U32.AND P0, PT, R0, c[0x0][0x148], PT
 @P0 EXIT
 ISCADD R2.CC, R0, c[0x0][0x140], 0x2
 MOV32I R3, 0x4
 MOV R4, c[0x0][0x148]
 IMAD.U32.U32.HI.X R3, R0, R3, c[0x0][0x144]
 ST.E [R2], R4
 EXIT
.L_1:
 BRA `(.L_1)
.L_22:
```

```
__global__ void test(int* out, int N)
{
    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < N)
    {
        out[tid] = N;
    }
}
```

28

.CC condition code register, single-bit register to hold carry flag
store uses register pair (R2, R3) for 64-bit address
IMAD.U32.U32.HI.X R3, R0, R3, c[0x0][0x144] computes upper 32-bit for address to store to

Example 2

```
__global__ void test(int* out, int N)
{
    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < N)
    {
        if (tid % 2 == 0)
            out[tid] = out[tid] + 2;
        else
            out[tid] = out[tid] - 8;
    }
}
```

https://godbolt.org/z/4vT1bK

29

Example 2: Predication

```
test(int*, int):
 MOV R1, c[0x0][0x44]
 S2R R0, SR_CTAID.X
 S2R R3, SR_TID.X
 IMAD R0, R0, c[0x0][0x28], R3
 ISETP.GE.U32.AND P0, PT, R0, c[0x0][0x148], PT
 @P0 EXIT
 ISCADD R2.CC, R0, c[0x0][0x140], 0x2
 MOV32I R3, 0x4
 IMAD.U32.U32.HI.X R3, R0, R3, c[0x0][0x144]
 LOP32I.AND R0, R0, 0x1
 LD.E R4, [R2]
 ISETP.NE.U32.AND P0, PT, R0, 0x1, PT
 @!P0 IADD32I R5, R4, -0x8
 @!P0 ST.E [R2], R5
 @!P0 EXIT
 IADD32I R4, R4, 0x2
 ST.E [R2], R4
 EXIT
.L_1:
 BRA `(.L_1)
.L_22:
```

```
__global__ void test(int* out, int N)
{
    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < N)
    {
        if (tid % 2 == 0)
            out[tid] = out[tid] + 2;
        else
            out[tid] = out[tid] - 8;
    }
}
```

30

Example 3

```
__device__ void A();
__device__ void B();
__device__ void C();

__global__ void test()
{
    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < 16)
    {
        A();
    }
    else
    {
        B();
    }

    C();
}
```

https://godbolt.org/z/qPY64h

Example 3: Branch Stack

```
0xFFFFFFFF  →  test():
0xFFFFFFFF  →   MOV R1, c[0x0][0x44]
0xFFFFFFFF  →   S2R R0, SR_CTAID.X
0xFFFFFFFF  →   SSY `(.L_2)
0xFFFFFFFF  →   S2R R3, SR_TID.X
0xFFFFFFFF  →   IMAD R0, R0, c[0x0][0x28], R3
0xFFFFFFFF  →   ISETP.GE.U32.AND P0, PT, R0, 0x10, PT
0xFFFFFFFF  →   @!P0 BRA `(.L_3)
0xFFFF0000  →   JCAL `(_Z1Bv)
0xFFFF0000  →   NOP.S
0x0000FFFF  →  .L_3:
0x0000FFFF  →   JCAL `(_Z1Av)
0x0000FFFF  →   NOP.S
0xFFFFFFFF  →  .L_2:
0xFFFFFFFF  →   JCAL `(_Z1Cv)
0xFFFFFFFF  →   MOV RZ, RZ
0xFFFFFFFF  →   EXIT
```

Branch Stack

mask	PC
0xFFFFFFFF	.L_2
0x0000FFFF	.L_3

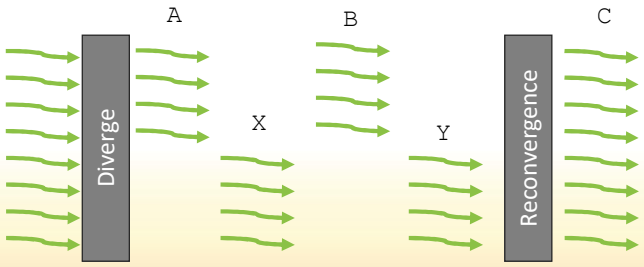
```
__global__ void test()
{
    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < 16)
    {
        A();
    }
    else
    {
        B();
    }

    C();
}
```

32

Independent Thread Scheduling

- program counter per thread
- schedule branches instead of warps
  - forward-progress guarantee for all branches
- better utilization: one branch not ready → the other one might be

```
if(threadIdx.x & 0x4)
{
    A();
    X();
}
else
{
    B();
    Y();
}
__warpsync();
C();
```

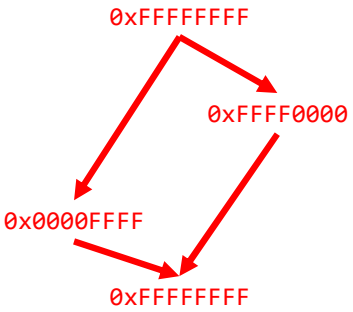


Example 3: Independent Thread Scheduling

no sync needed to switch to other branch

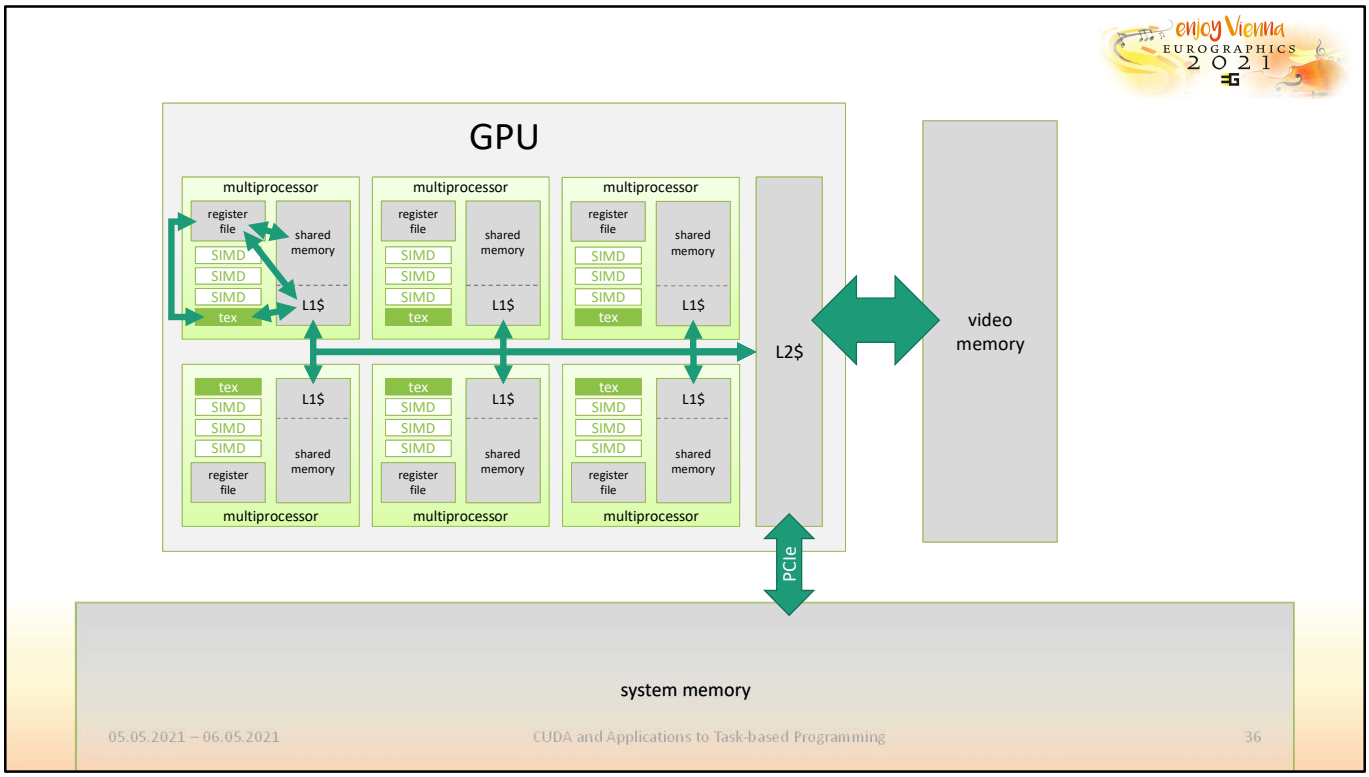
```
test():
 IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28]
 S2R R0, SR_CTAID.X
 BMOV.32.CLEAR RZ, B6
 BSSY B6, `(.L_5)
 S2R R3, SR_TID.X
 IMAD R0, R0, c[0x0][0x0], R3
 ISETP.GE.U32.AND P0, PT, R0, 0x10, PT
 @!P0 BRA `(.L_6)
 MOV R20, 32@lo((test() + .L_2@srel))
 MOV R21, 32@hi((test() + .L_2@srel))
 CALL.ABS.NOINC `(_Z1Bv)
.L_2:
 BRA `(.L_3)
.L_6:
 MOV R20, 32@lo((test() + .L_3@srel))
 MOV R21, 32@hi((test() + .L_3@srel))
 CALL.ABS.NOINC `(_Z1Av)
.L_3:
 BSYNC B6
.L_5:
 MOV R20, 32@lo((test() + .L_4@srel))
 MOV R21, 32@hi((test() + .L_4@srel))
 CALL.ABS.NOINC `(_Z1Cv)
.L_4:
 EXIT
.L_7:
 BRA `(.L_7)
.L_22:
```

Branch Tree



The Memory Hierarchy

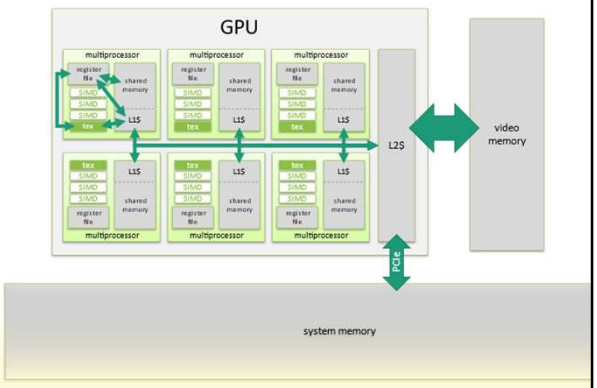
Memory tends to be *the* bottleneck in any modern system. And GPUs are no exception. In fact, given the massive amount of parallel computation that is typically taking place on a GPU at any given moment, the effect of memory on performance is arguably even more pronounced on the GPU than we might be used to from working on the CPU. Thus, in many ways, GPU programming really is all about making the most of the GPU's memory subsystem.


GPU
multiprocessor
register file
shared memory
SIMD
tex
L1$
L2$
video memory
PCIe
system memory
05.05.2021 – 06.05.2021
CUDA and Applications to Task-based Programming
36

CUDA Memory Spaces

- multiple paths to fetch from memory
- different trade-offs
- special-purpose hardware

- exposed in CUDA in the form of memory spaces



CUDA Memory Spaces Overview

- Global Memory
  - shared by all threads on the device
  - read and write
  - cached (L2$ and L1$)
  - general-purpose data store
- Local Memory
  - private to each thread
  - register spills, stack
  - read and write
  - cached (L2$ and L1$)
- Registers
  - private to each thread
- Shared Memory
  - shared by threads within the same block
  - low-latency communication
- Texture Memory
  - read-only
  - spatially-local access
  - hardware filtering, format conversion, border handling
- Constant Memory
  - read-only
  - optimized for broadcast access

38

In CUDA, these hardware resources are exposed in the form of a number of memory spaces, each with different properties designed for different kinds of access patterns. CUDA applications take advantage of the various capabilities found within the GPU's memory system by placing and accessing data in the corresponding CUDA memory space.

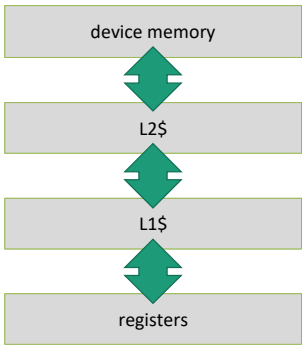
We will now have a more detailed look at each one of these memory spaces and what they have to offer.

Global Memory

- general-purpose data store
- backed by device memory
  - use for input and output data
  - linear arrays
- relatively slow
  - bandwidth: ≈ 300–700 GiB/s (GDDR5/6 vs HMB2)
  - non-cached coalesced access: 375 cycles
  - L2 cached access: 190 cycles
  - L1 caches access: 30 cycles

⇒ crucial to utilize caches

⇒ access pattern important



39

The most important memory space is global memory. It corresponds to device memory and is accessible to all threads running on the GPU. As a result of this wide scope, memory accesses potentially have to bubble all the way up to, or all the way down from device memory. Making use of the available caches is, thus, vital for performance.

Global Memory: Caches

- purpose not the same as CPU caches
  - much smaller size (especially per thread)
  - goal is not minimizing individual access latency via temporal reuse
  - instead: smooth-out access patterns
- don't try cache blocking like on the CPU
  - 100s of threads accessing L1$
  - 1000s of threads accessing L2$
  - use shared memory instead

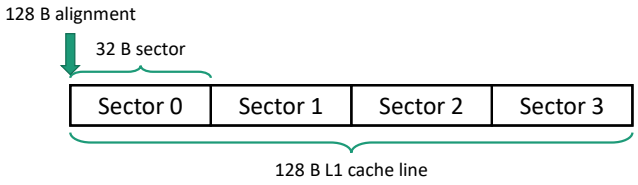
Example:
68 SMs
2048 threads per SM
5120 KiB of L2$
128 KiB of L1$
→ 64 B L1$, 37 B L2$ per thread

40

If we look at a current high-end gaming GPU with 68 multiprocessors, up to 2048 resident threads per multiprocessor, 5120 KiB of L2 cache, and 128 KiB of L1 cache, we are left with 64 Bytes of L1 and about 37 Bytes of L2 cache per thread.

Global Memory: Transactions

- memory access granularity / cacheline size
  - L1$ / L2$: 32 B / 128 B (4 sectors of 32 B)
- stores
  - write-through for L1$
  - write-back for L2$
- memory operations issued per warp
  - threads provide addresses
  - combined to lines/segments needed
  - requested and served
- try to get coalescing per warp
  - align starting address
  - access within a contiguous region
  - ideal: consecutive threads access consecutive memory locations



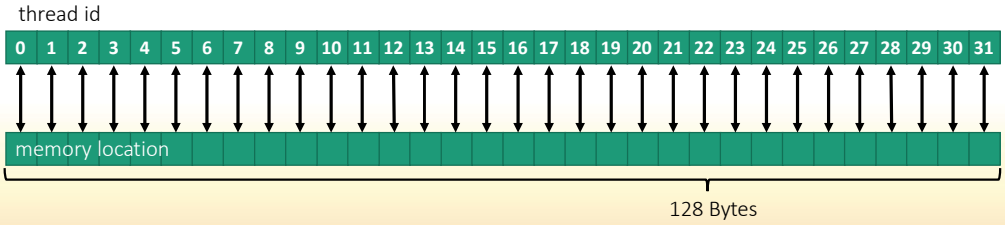
41

One cacheline is 128 Bytes, which is split into 4 sectors of Size 32 Bytes. Memory Transactions are 32 Byte long and only actually requested 32 Byte sectors are read from memory.

Granularity Example 1/4

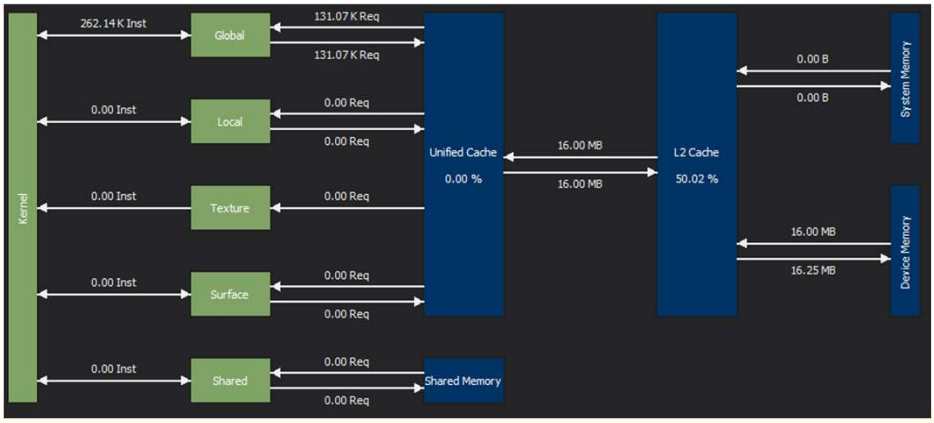
```
__global__ void testCoalesced(int* in, int* out, int elements)
{
  int block_offset = blockIdx.x*blockDim.x;
  int warp_offset = 32 * (threadIdx.x / 32);
  int laneid = threadIdx.x % 32;
  int id = (block_offset + warp_offset + laneid) % elements;

  out[id] = in[id];
}
```

testCoalesced done in 0.065568 ms ⇔ 255.875 GiB/s



42

Granularity Example 1/4

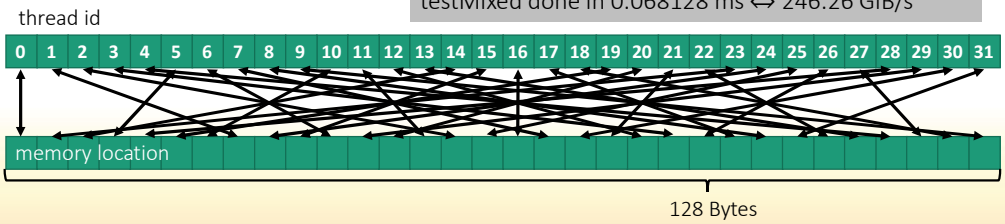


Coalesced

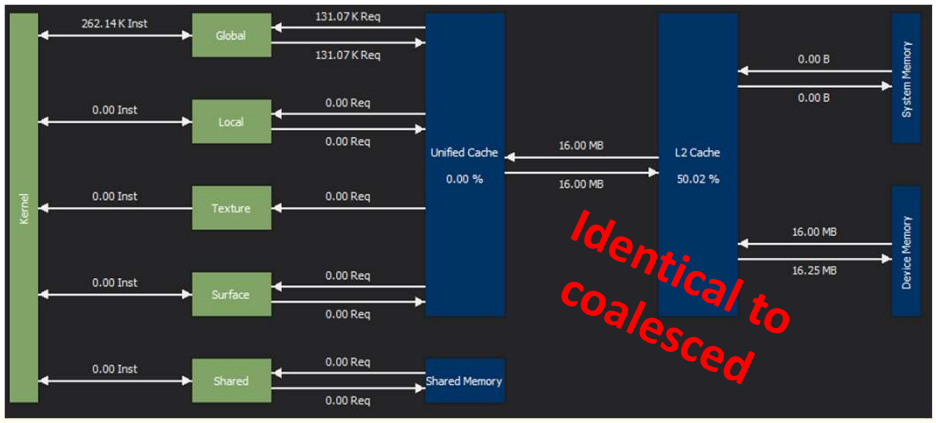
Granularity Example 2/4

```
__global__ void testMixed(int* in, int* out, int elements)
{
  int block_offset = blockIdx.x*blockDim.x;
  int warp_offset = 32 * (threadIdx.x / 32);
  int elementid = (threadIdx.x * 7) % 32;
  int id = (block_offset + warp_offset + elementid) % elements;

  out[id] = in[id];
}
```

testCoalesced done in 0.065568 ms ⇔ 255.875 GiB/s
testMixed done in 0.068128 ms ⇔ 246.26 GiB/s



Granularity Example 2/4

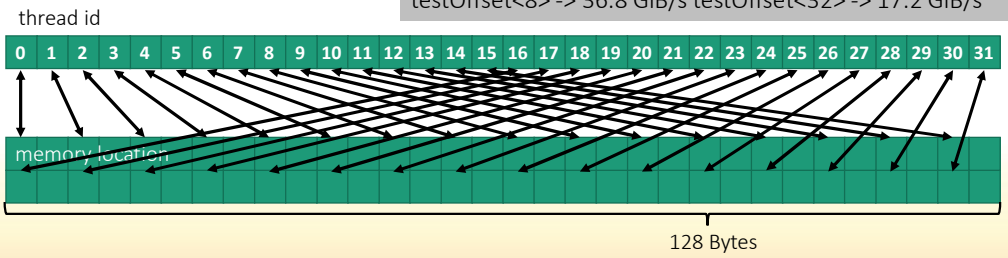


Mixed

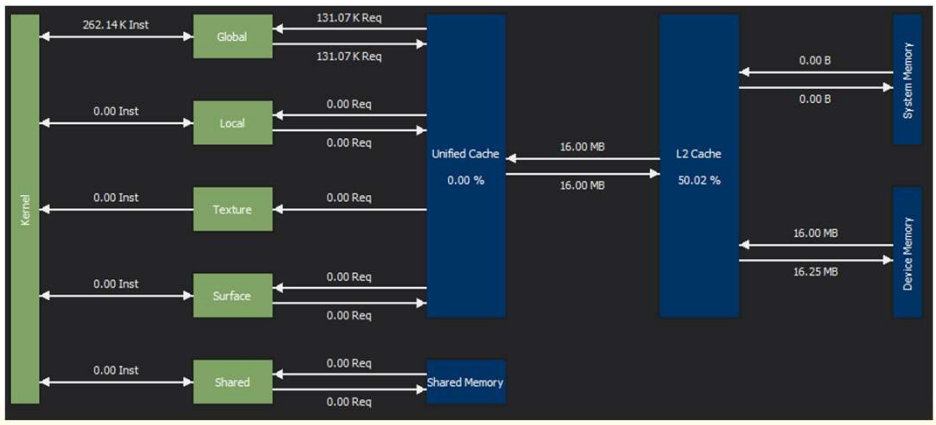
Granularity Example 3/4

```
template<int offset>
__global__ void testOffset(int* in, int* out, int elements)
{
  int block_offset = blockIdx.x*blockDim.x;
  int warp_offset = 32 * (threadIdx.x / 32);
  int laneid = threadIdx.x % 32;
  int id = ((block_offset + warp_offset + laneid) * offset) % elements;

  out[id] = in[id];
}
```

testCoalesced -> 255.875 GiB/s
testOffset<2> -> 134 GiB/s testOffset<4> -> 72.9 GiB/s
testOffset<8> -> 36.8 GiB/s testOffset<32> -> 17.2 GiB/s

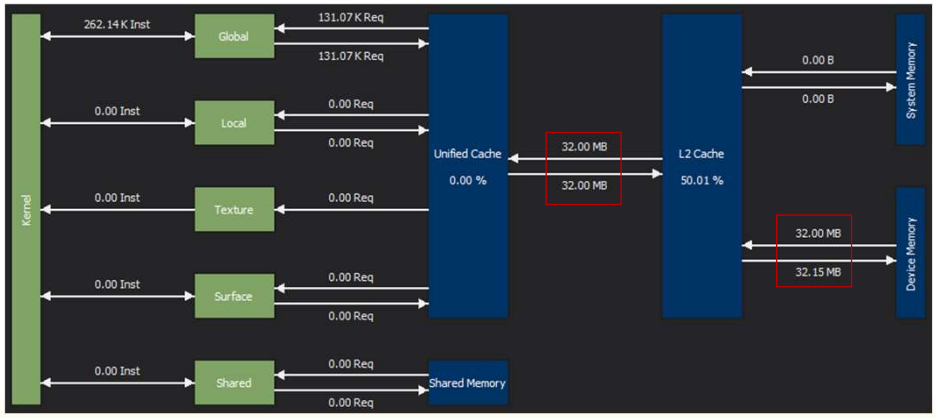


Granularity Example 3/4



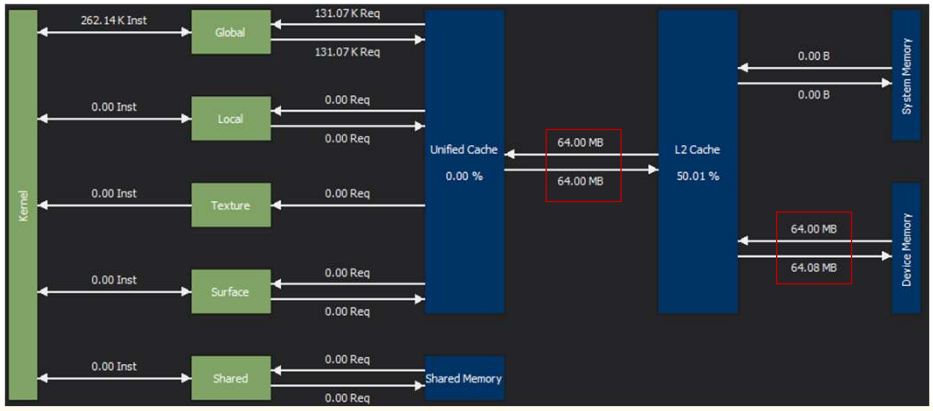
Coalesced 255.875 GB/s

Granularity Example 3/4



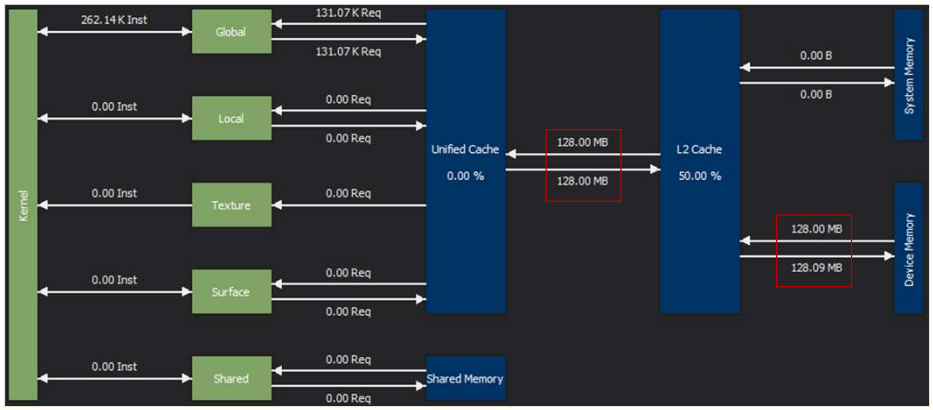
Offset<2> 134.123 GiB/s

Granularity Example 3/4



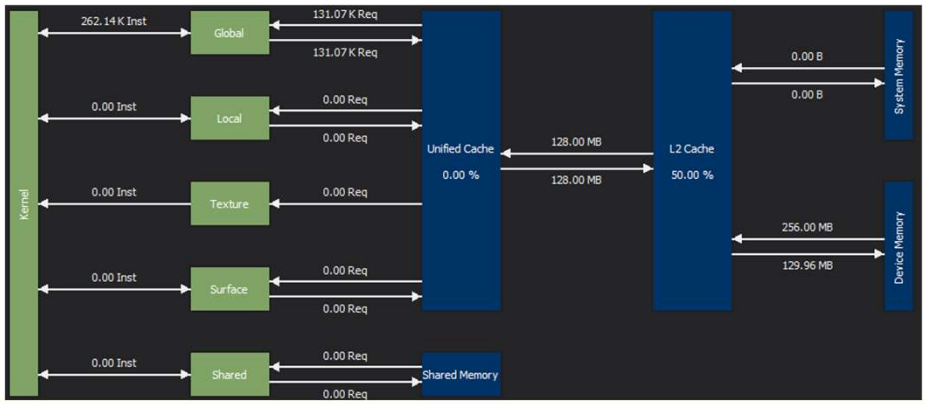
Offset<4> 72.9 GiB/s

Granularity Example 3/4



Offset<8> 36.81 GiB/s

Granularity Example 3/4



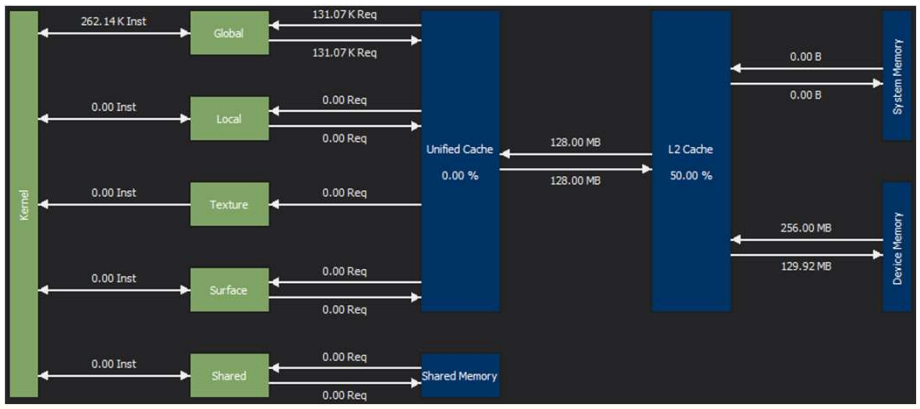
Offset<32> 17.26 GiB/s

Granularity Example 4/4

```
__global__ void testScattered(int* in, int* out, int elements)
{
  int block_offset = blockIdx.x*blockDim.x;
  int warp_offset = 32 * (threadIdx.x / 32);
  int elementid = threadIdx.x % 32;
  int id = ((block_offset + warp_offset + elementid) * 121) % elements;

  out[id] = in[id];
}
```

testCoalesced done in 0.0655 ms ⇔ 255.875 GiB/s
testOffset<32> done in 0.972 ms ⇔ 17.2605 GiB/s
testScattered done in 2.0385 ms ⇔ 8.23 GiB/s

Granularity Example 4/4



Scattered 8.23 GiB/s

Granularity Example Summary

- testCoalesced 0.066 ms ⇔ 255.875 GiB/s
  testMixed 0.068 ms ⇔ 246.260 GiB/s
  testOffset<2> 0.125 ms ⇔ 134.123 GiB/s
  testOffset<4> 0.230 ms ⇔ 72.979 GiB/s
  testOffset<8> 0.456 ms ⇔ 36.812 GiB/s
  testOffset<32> 0.972 ms ⇔ 17.260 GiB/s
  testScattered 2.039 ms ⇔ 8.230 GiB/s

- access pattern within 128 Byte segment does not matter
- offset between data → more requests need to be handled
- peak performance not met due to computation overhead
- more scattered data access slower with GDDR RAM

Vector Loads / Stores

- many kernels bandwidth bound
  - ever copy operation consists of four steps
    - compute load & store address (IMAD)
    - load (LD) & store (ST)

```
__global__ void device_copy_scalar_kernel(int* d_in, int* d_out, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = idx; i < N; i += blockDim.x * gridDim.x) {
      d_out[i] = d_in[i];
    }
}
```

```
/*0058*/ IMAD R6.CC, R0, R9, c[0x0][0x140]
/*0060*/ IMAD.HI.X R7, R0, R9, c[0x0][0x144]
/*0068*/ IMAD R4.CC, R0, R9, c[0x0][0x148]
/*0070*/ LD.E R2, [R6]
/*0078*/ IMAD.HI.X R5, R0, R9, c[0x0][0x14c]
/*0090*/ ST.E [R4], R2
```


```
__global__ void copy(int* __restrict d_out, const int* __restrict d_in, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    for (int i = idx; i < N; i += blockDim.x * gridDim.x)
        d_out[i] = d_in[i];
}
```


```
__global__ void copy(int* __restrict d_out, const int* __restrict d_in, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    for (int i = idx; i < N; i += blockDim.x * gridDim.x)
        d_out[i] = d_in[i];
}
```

```
copy(int*, int const*, int):
 MOV R1, c[0x0][0x28]
 S2R R0, SR_CTAID.X
 S2R R3, SR_TID.X
 IMAD R0, R0, c[0x0][0x0], R3
 ISETP.GE.AND P0, PT, R0, c[0x0][0x170], PT
 @P0 EXIT
.L_1:
 MOV R5, 0x4
 IMAD.WIDE R2, R0, R5, c[0x0][0x168]
 LDG.E.CONSTANT.SYS R3, [R2]
 IMAD.WIDE R4, R0, R5, c[0x0][0x160]
 MOV R7, c[0x0][0xc]
 IMAD R0, R7, c[0x0][0x0], R0
 ISETP.GE.AND P0, PT, R0, c[0x0][0x170], PT
 STG.E.SYS [R4], R3
 @!P0 BRA `(.L_1)
 EXIT
.L_2:
 BRA `(.L_2)
.L_25:
```

address to load from

load

address to store to

store

57


```
__global__ void copy4(int4* __restrict d_out, const int4* __restrict d_in, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    for (int i = idx; i < N / 4; i += blockDim.x * gridDim.x)
        d_out[i] = d_in[i];
}
```

address to load from
load
address to store to

store

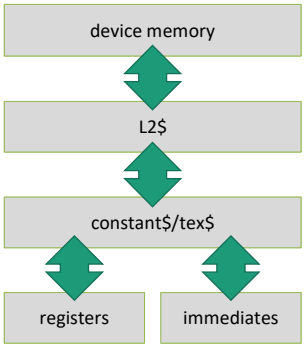
```
copy4(int4*, int4 const*, int):
 MOV R1, c[0x0][0x28]
 S2R R0, SR_CTAID.X
 ULDC UR4, c[0x0][0x170]
 USHF.R.S32.HI UR4, URZ, 0x1f, UR4
 S2R R3, SR_TID.X
 ULDC UR5, c[0x0][0x170]
 ULEA.HI UR4, UR4, UR5, URZ, 0x2
 USHF.R.S32.HI UR4, URZ, 0x2, UR4
 IMAD R0, R0, c[0x0][0x0], R3
 ISETP.GE.AND P0, PT, R0, UR4, PT
 @P0 EXIT
 BMOV.32.CLEAR RZ, B0
 BSSY B0, `(.L_1)
.L_2:
 MOV R3, 0x10
 IMAD.WIDE R4, R0, R3, c[0x0][0x168]
 LDG.E.128.CONSTANT.SYS R4, [R4]
 IMAD.WIDE R2, R0, R3, c[0x0][0x160]
 MOV R9, c[0x0][0xc]
 IMAD R0, R9, c[0x0][0x0], R0
 ISETP.GE.AND P0, PT, R0, UR4, PT
 STG.E.128.SYS [R2], R4
 @!P0 BRA `(.L_2)
 BSYNC B0
.L_1:
 EXIT
.L_3:
```

Vector Loads / Stores

- improve performance by using vectorized loads and stores
  - use vector data types (e.g. `int2`, `float4`, `uchar4`, …)
    - require aligned data
  - still 6 instructions
    - but loads 2×/4× more data
- can help alleviate impact of suboptimal access pattern
- but slightly increase register pressure
  - more data at once requires more registers to hold

Constant Memory

- read-only
- ideal for data read uniformly by warps
  - e.g.: coefficients, used to pass kernel parameters
- supports broadcasting
  - all threads read same value -> data broadcasted to all threads simultaneously
  - otherwise diverged -> slowdown
- limited to 64 KiB



Constant Memory

```
__constant__ float myarray[128];
__global__ void kernel()
{
  …
  float x = myarray[23];               //uniform
  float y = myarray[blockIdx.x + 2];   //uniform
  float z = myarray[threadIdx.x];      //non-uniform
  …
}
```

Constant Memory: Example

```
__constant__ float c_array[128];
__global__ void kernel(float* __restrict d_array, const float* __restrict dc_array)
{
  float a = c_array[0];              // const cache  24cyc
  float b = c_array[blockIdx.x];     // const cache  24cyc
  float c = c_array[threadIdx.x];    // const cache  35cyc
  float d = d_array[blockIdx.x];     // L2 cache     24cyc
  float e = d_array[threadIdx.x];    // L2 cache     26cyc
  float f = dc_array[blockIdx.x];    // L1 cache     23cyc
  float g = dc_array[threadIdx.x];   // L1 cache     24cyc
}
```

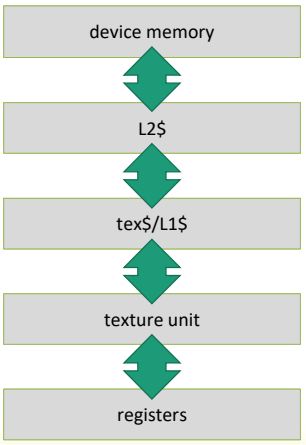
https://godbolt.org/z/sc1GGTfse

Constant Memory: Conclusion

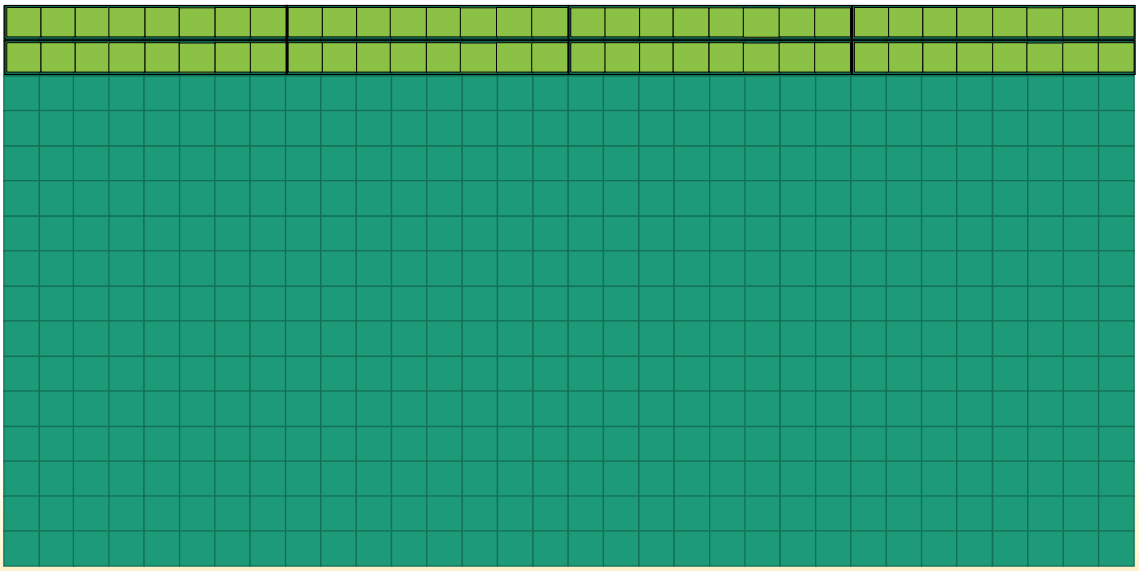
- only fast if all threads within a warp read the same value
- can be faster than global
- uses different cache hierarchy than global
- compiler can automatically fetch things through constant memory
  - can also be done manually using `__ldg()` intrinsics

Texture Memory

- `cudaArray`: image data laid out in memory to optimize data locality for spatially-local access
  - e.g., accesses within 2D region should hit the cache
- Texture Object
  - fetch from `cudaArray` or Global Memory
  - filtering, border handling, format conversion
  - read-only
- Surface Object
  - read/write `cudaArrays`
  - concurrent writing and reading as texture: undefined result



RowAfterRow - 1D



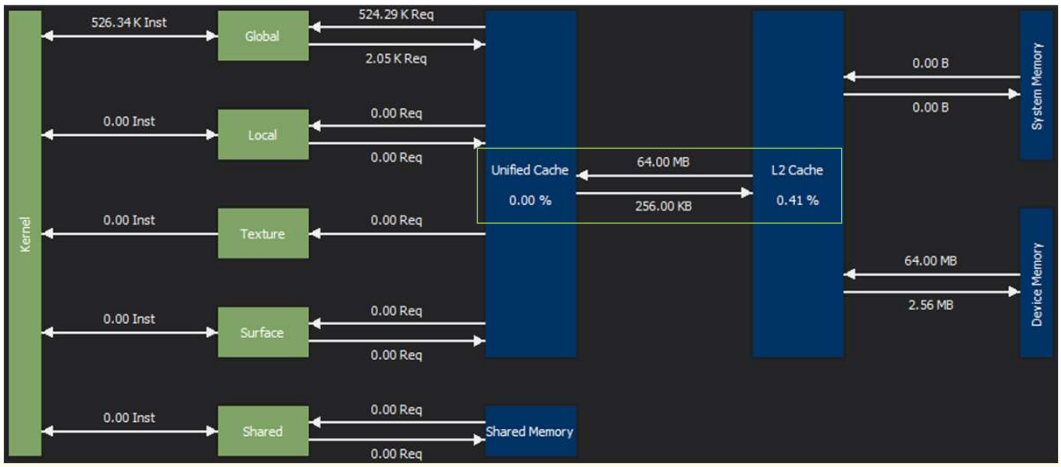
RowAfterRow: Texture vs Global

```
__global__ void deviceLoadRowAfterRow(const uchar4* data, int width, int height, uchar4* out)
{
    uchar4 sum = make_uchar4(0,0,0,0);

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int rowid{0}, colid{0};

    while(tid < (height * width))
    {
        rowid = tid / width;
        colid = tid % width;
        uchar4 in = data[rowid * width + colid];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
        tid += (blockDim.x * gridDim.x);
    }
    out[blockIdx.x*blockDim.x + threadIdx.x] = sum;
}
```

const uchar4* __restrict data

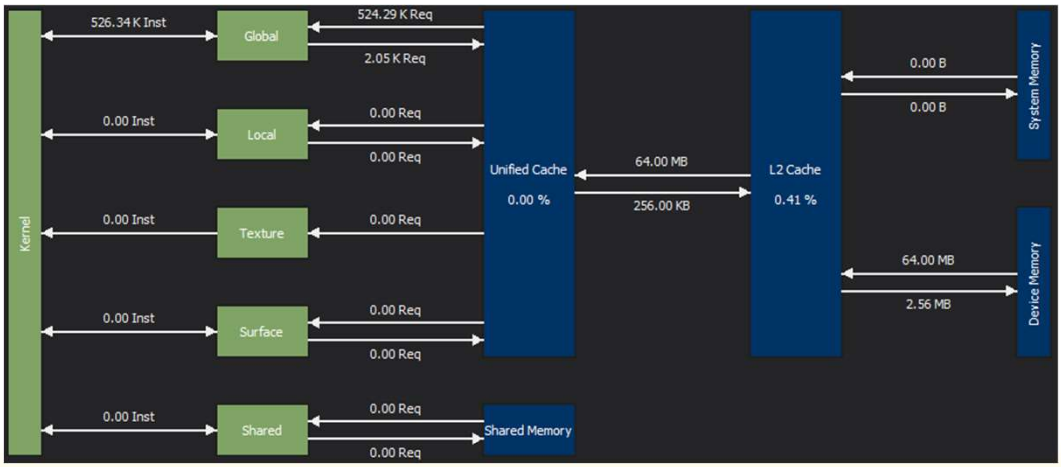
uchar4 in = tex2D(myTex,colid,rowid);

RowAfterRow: Texture vs Global



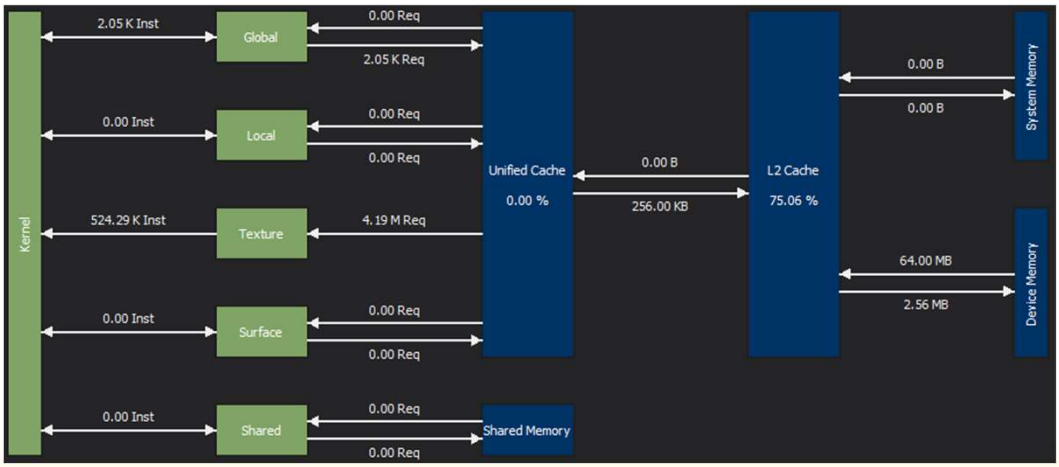
global linear (cudaMalloc) 445 GiB/s

RowAfterRow: Texture vs Global



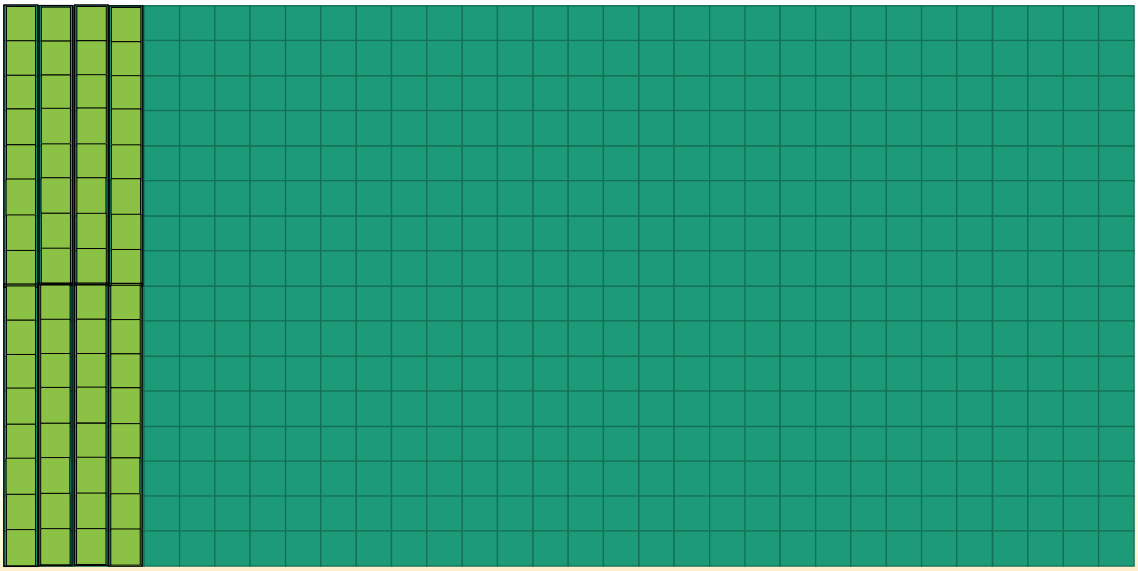
global restrict (cudaMalloc) 464 GiB/s

RowAfterRow: Texture vs Global



texture (cudaArray) 387 GiB/s

ColumnAfterColumn - 1D



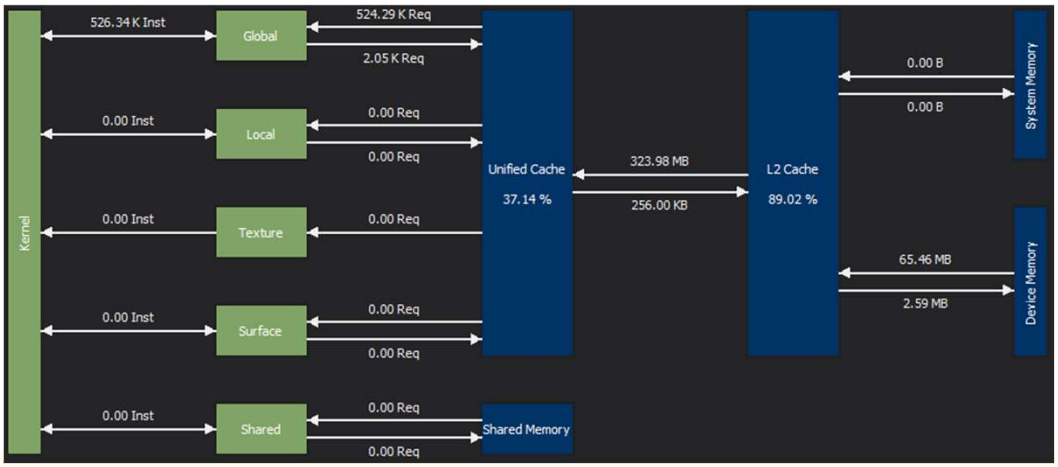
ColumnAfterColumn: Texture vs Global

```
__global__ void deviceLoadColumnAfterColumn(const uchar4* data, int width, int height, uchar4* out)
{
    uchar4 sum = make_uchar4(0,0,0,0);

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int rowid{0}, colid{0};

    while(tid < (height * width))
    {
        rowid = tid % height;
        colid = tid / height;
        uchar4 in = data[rowid * width + colid];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
        tid += (blockDim.x * gridDim.x);
    }
    out[blockIdx.x*blockDim.x + threadIdx.x] = sum;
}
```

```
const uchar4* __restrict data
```

```
uchar4 in = tex2D(myTex,colid,rowid);
```

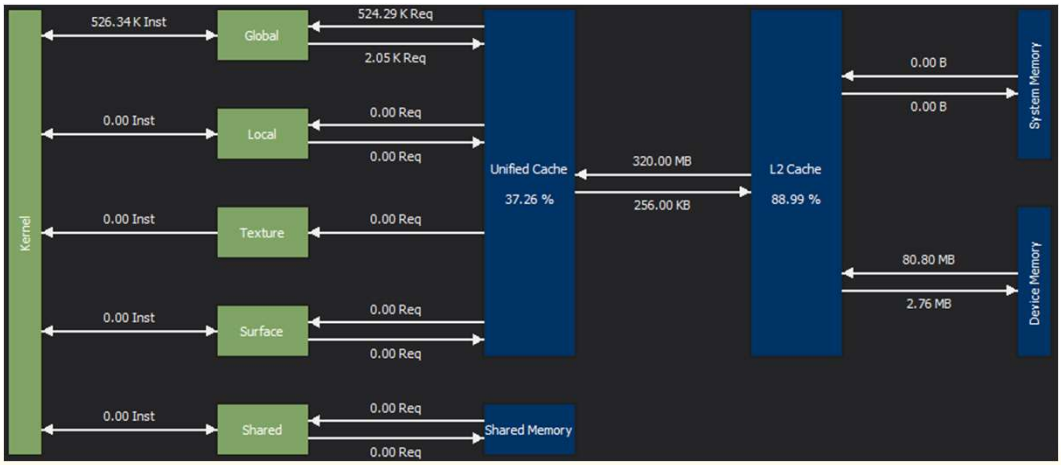
ColumnAfterColumn: Texture vs Global



global linear (cudaMalloc) 268 GiB/s

was 445 GiB/s

ColumnAfterColumn: Texture vs Global

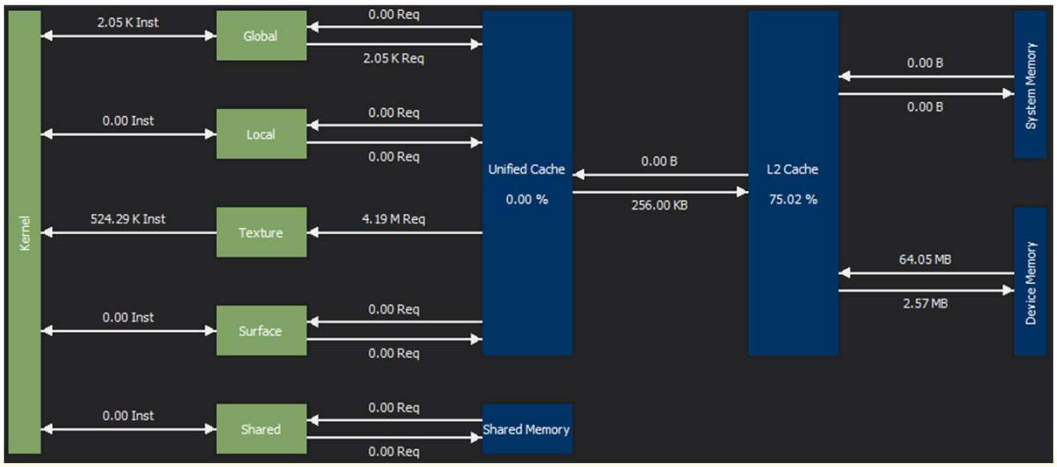


global restrict (cudaMalloc) 270 GiB/s

was 464 GiB/s

73

ColumnAfterColumn: Texture vs Global



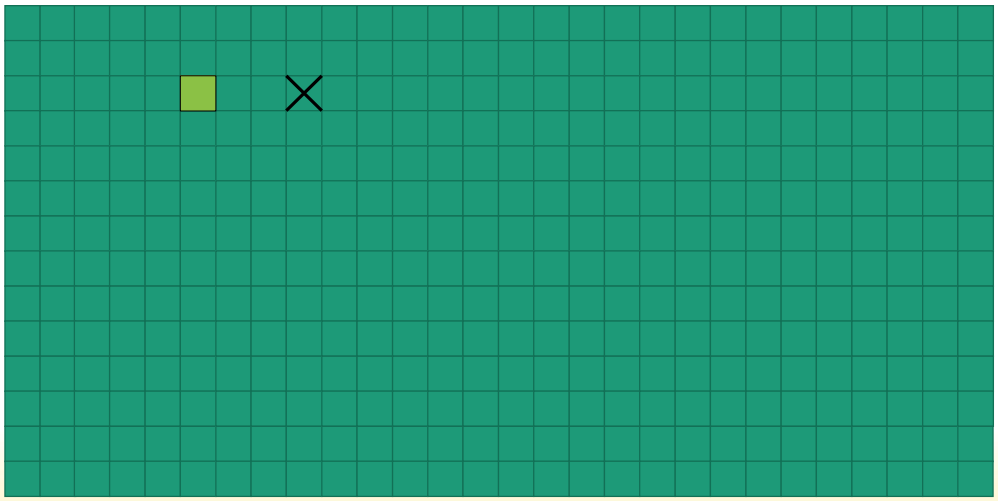
texture (cudaArray) 368 GiB/s

was 387 GiB/s

Texture vs Global: 1D Access

- row access is similarly efficient with all types of memory
  - virtually no cache usage
  - slighty better with linear memory layout vs textures
- column access significantly slower for global vs texture
  - texture only slightly slower than row access

Filter X



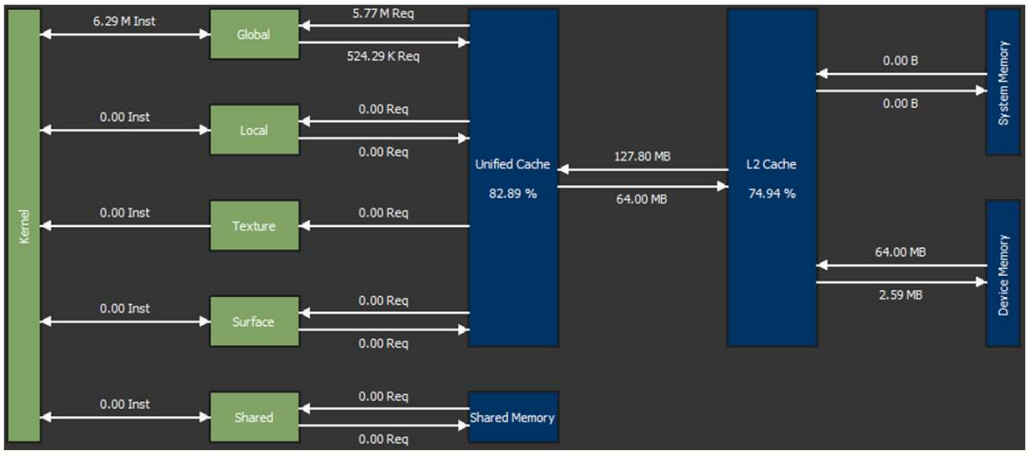
FilterX: Texture vs Global

```
__global__ void globalFilterX(const uchar4* data, int pitch, int width, int height,
uchar4* out, int offset)
{
  uchar4 sum = make_uchar4(0,0,0,0);
  int xin = blockIdx.x*blockDim.x + threadIdx.x;
  int yin = blockIdx.y*blockDim.y + threadIdx.y;

  if(xin >= offset && xin < width-offset-1)
  {
      for(int x = xin-offset; x <= xin+offset; ++x)
      {
        uchar4 in = data[x + yin*pitch];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
      }
  }
  yin = yin % blockDim.y;
  out[xin + yin*gridDim.x*blockDim.x] = sum;
}
```

No need for if

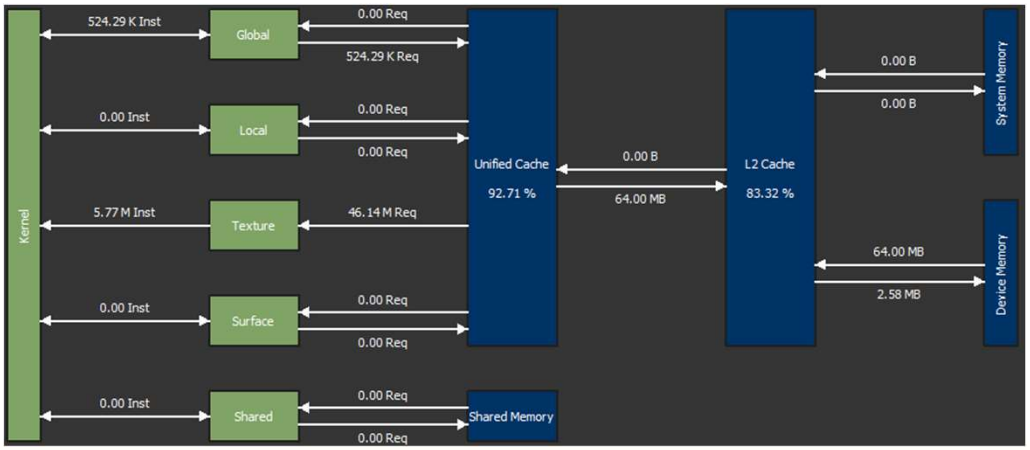
```
uchar4 in = tex2D(myTex,x,yin);
```

FilterX: Texture vs Global



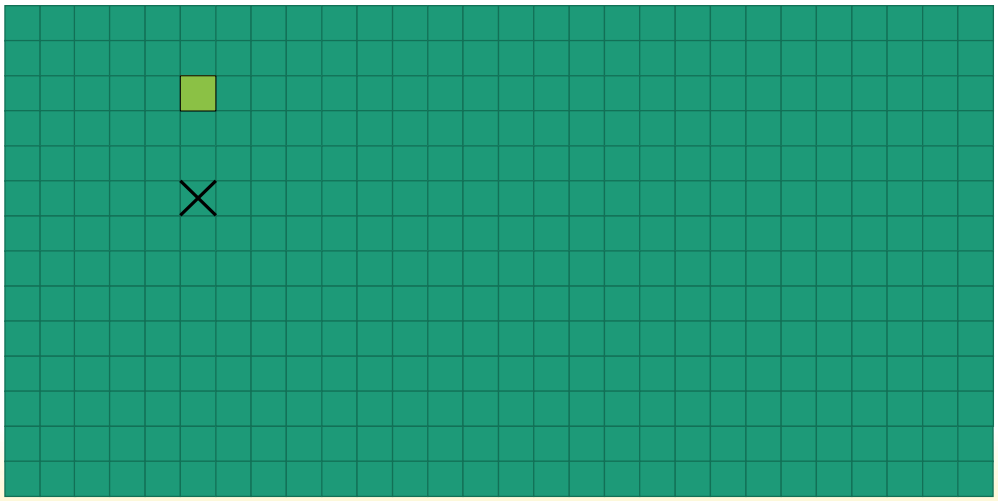
global linear (cudaMalloc) 1602 GiB/s

FilterX: Texture vs Global



texture (cudaArray) 1386 GiB/s

Filter Y

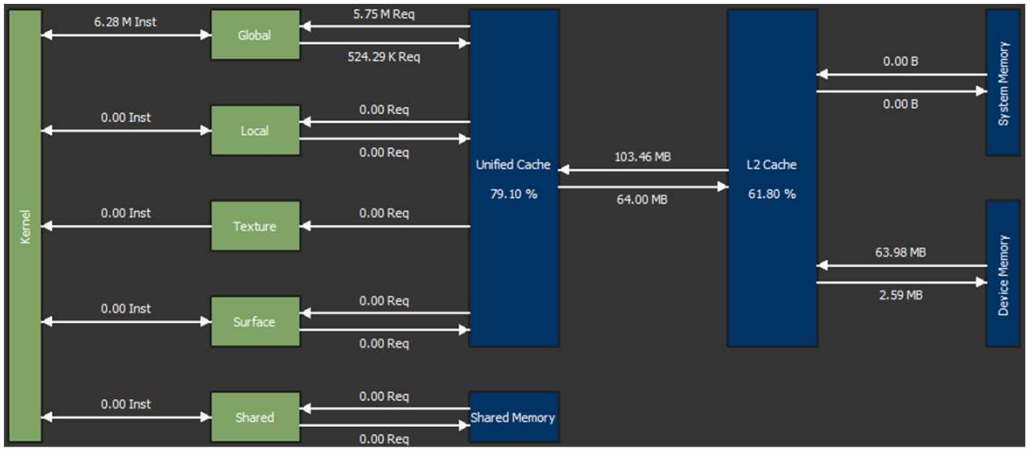


FilterY: Texture vs Global

```
__global__ void globalFilterX(const uchar4* data, int pitch, int width, int height,
uchar4* out, int offset)
{
  uchar4 sum = make_uchar4(0,0,0,0);
  int xin = blockIdx.x*blockDim.x + threadIdx.x;
  int yin = blockIdx.y*blockDim.y + threadIdx.y;

  if(yin >= offset && yin < height-offset-1) No need for if
  {
      for(int y = yin-offset; y <= yin+offset; ++y)
      {
        uchar4 in = data[xin + y*pitch];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
      }
  }
  yin = yin % blockDim.y;
  out[xin + yin*gridDim.x*blockDim.x] = sum;
}
```

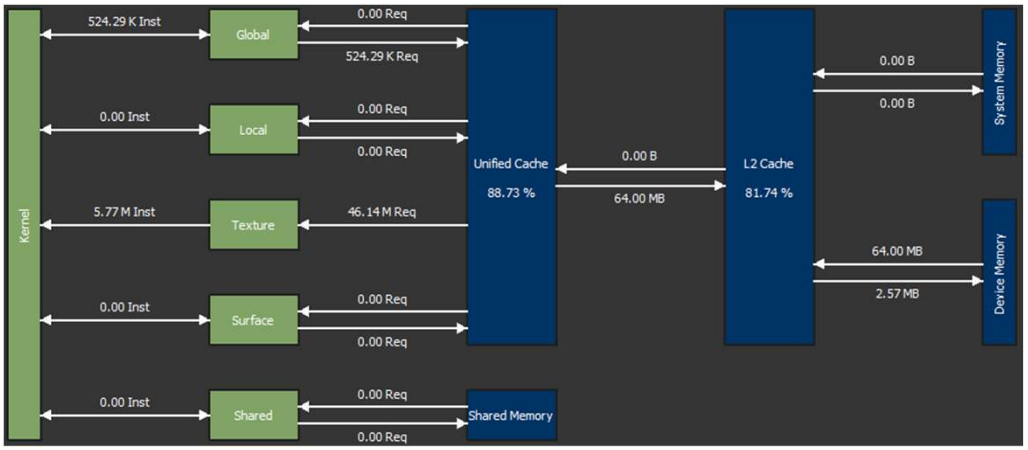
```
uchar4 in = tex2D(myTex,xin,y);
```

FilterY: Texture vs Global



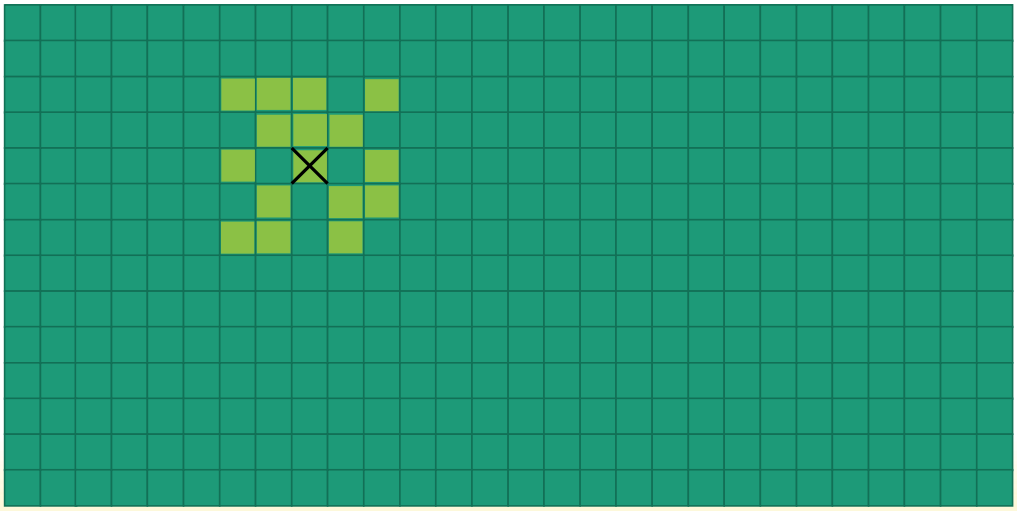
global linear (cudaMalloc) 1560GB/s

FilterY: Texture vs Global



texture (cudaArray) 1411 GiB/s

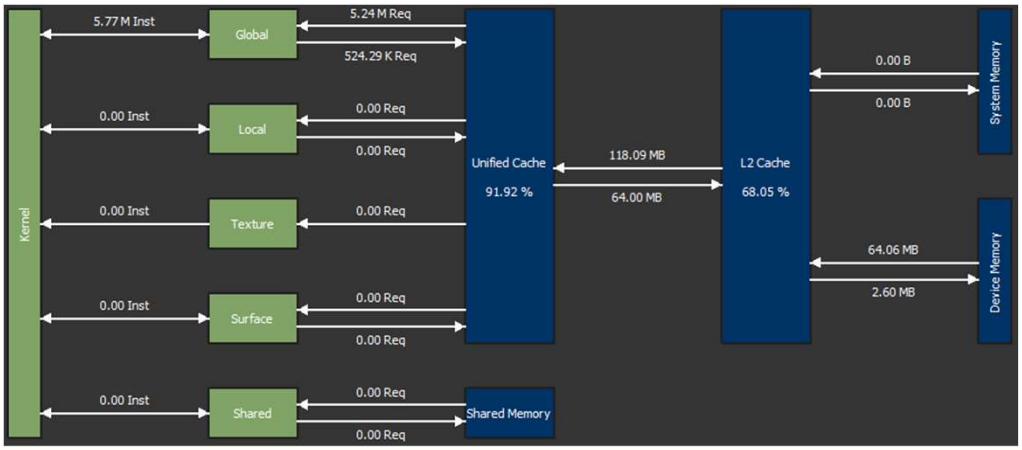
Random Access in vicinity



Random Sampling: Texture vs Global

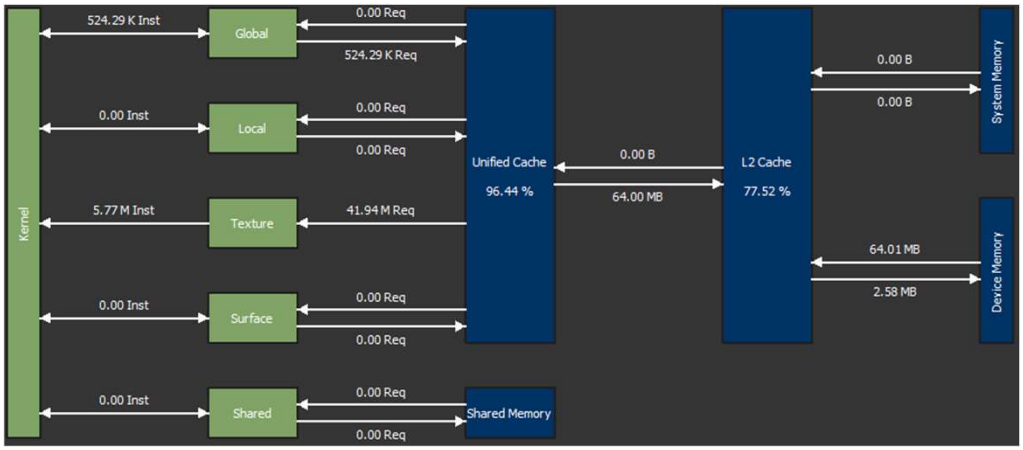
```
template<int Area>
__global__ void deviceReadRandom(const uchar4* data, int pitch, int width, int height, uchar4* out, int samples){
  uchar4 sum = make_uchar4(0,0,0,0);
  int xin = blockIdx.x*blockDim.x + Area*(threadIdx.x/Area);
  int yin = blockIdx.y*blockDim.y + Area*(threadIdx.y/Area);

  unsigned int xseed = threadIdx.x *9182  + threadIdx.y*91882  + threadIdx.x*threadIdx.y*811 + 72923181;
  unsigned int yseed = threadIdx.x *981   + threadIdx.y*124523 + threadIdx.x*threadIdx.y*327 + 98721121;

  for(int sample = 0; sample < samples; ++sample)
  {
    unsigned int x = xseed%Area;
    unsigned int y = yseed%Area;
    xseed = (xseed * 1587);
    yseed = (yseed * 6971);
    uchar4 in = data[xin + x + (yin + y)*pitch];
    sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
  }

  yin = (yin + threadIdx.y%Area) % blockDim.y;
  out[xin + threadIdx.x%Area + yin*width] = sum;
}
```

Random Sampling: Texture vs Global



global linear (cudaMalloc) 1014 GiB/s

Random Sampling: Texture vs Global



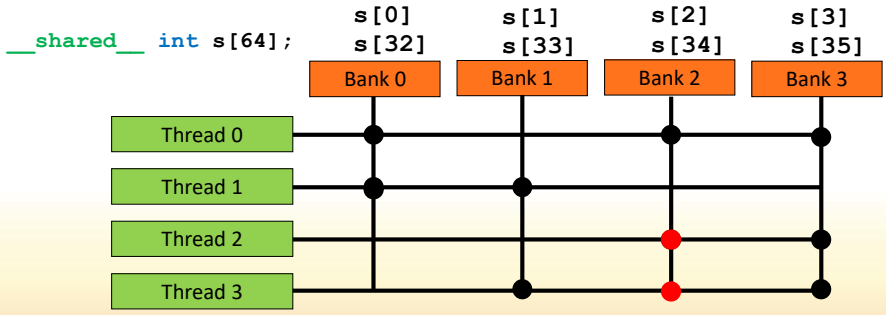
texture (cudaArray) 1170 GiB/s

Texture vs Global: Conclusion

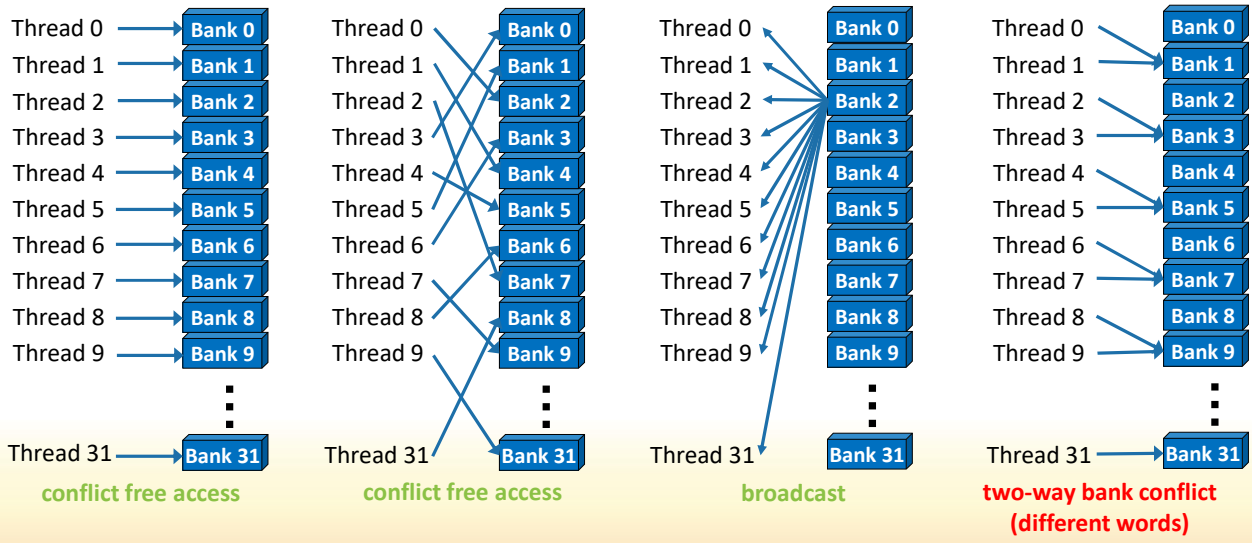
- prior to Volta
  - Textures tend to perform at least as good, sometimes better
  - put less stress on L2 cache
  - L1 cache free for other tasks
- now
  - advanced Unified Cache (L1 + Tex)
  - Textures still perform best for spatial access pattern
  - but can also be slower if access pattern and cache hits favor linear memory
- unique features: filtering, border handling, format conversion

Shared Memory

- shared access within one block (lifetime: block)
- located on multiprocessor → very fast
- limited size (32–96 KiB)
- crossbar: simultaneous access to distinct 32-Bit banks

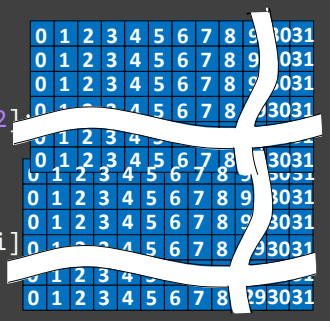


Shared Memory: Bank Conflicts



Shared Memory: Bank Conflicts

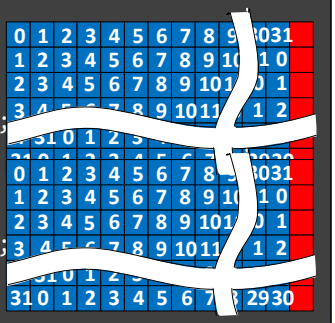
```
__global__ void kernel(...)
{
    __shared__ float mydata[32*32];
    ...
    float sum = 0;
    for(uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x + i*32];
    ...
    sum = 0;
    for(uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x*32 + i];
    ...
}
```



numbers represent banks

Shared Memory: Bank Conflict Resolution

```
__global__ void kernel(...)
{
    __shared__ float mydata[32*(32 + 1)];
    ...
    float sum = 0;
    for(uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x + i*33];
    ...
    sum = 0;
    for(uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x*33 + i];
    ...
}
```



Shared Memory: Use Cases

- inter-thread communication

```
__global__ void kernel(...)
{
    __shared__ bool run;
    run = true;
    while(run)
    {
        __syncthreads();
        if(found_it())
            run = false;
        __syncthreads();
    }
}
```

Shared Memory: Use Cases

- inter-thread communication
- reduce global memory access → manual cache

```
__global__ void kernel(float* global_data, ...)
{
    extern __shared__ float data[];
    uint linid = blockIdx.x*blockDim.x + threadIdx.x;
    //load
    data[threadIdx.x] = global_data[linid];
    __syncthreads();
    for(uint it = 0; it < max_it; ++it)
        calc_iteration(data); //calc
    __syncthreads();
    //write back
    global_data[linid] = data[threadIdx.x];
}
```

Shared Memory: Use Cases

- inter-thread communication
- reduce global memory access → manual cache
- adjust global memory access pattern

```
__global__ void transp(float* global_data, float* global_data2)
{
    extern __shared__ float data[];
    uint linid1 = blockIdx.x*32 + threadIdx.x + blockIdx.y*32*width;
    uint linid2 = blockIdx.x*32*width + threadIdx.x + blockIdx.y*32;
    for(uint i = 0; i < 32; ++i)
        data[threadIdx.x + i*33] = global_data[linid1 + i*width];
    __syncthreads();
    for(uint j = 0; j < 32; ++j)
        global_data2[linid2 + j*width] = data[threadIdx.x*33 + j] ;
}
```


95

Shared Memory: Use Cases

- inter-thread communication
- reduce global memory access → manual cache
- adjust global memory access pattern
- indexed access

```
__global__ void kernel(...)
{
    uint mydata[8]; //will be spilled to local memory
    for(uint i = 0; i < 8; ++i)
        mydata[i] = complexfunc(i, threadIdx.x);
    uint res = 0;
    for(uint i = 0; i < 64; ++i)
        res += secondfunc(mydata[(threadIdx.x + i) % 8],
                          mydata[i*threadIdx.x % 8]);
}
```

Shared Memory use cases

- inter-thread communication
- reduce global memory access → manual cache
- adjust global memory access pattern
- indexed access

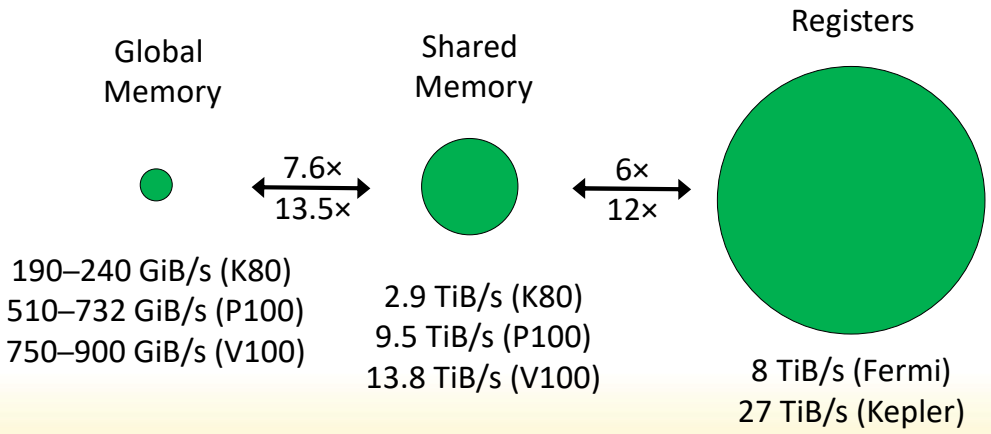
```
__global__ void kernel(...)
{
    __shared__ uint allmydata[8*BlockSize];
    uint *mydata = allmydata + 8*threadIdx.x;
    for(uint i = 0; i < 8; ++i)
        mydata[i] = complexfunc(i, threadIdx.x);
    uint res = 0;
    for(uint i = 0; i < 64; ++i)
        res += secondfunc(mydata[(threadIdx.x + i) % 8],
                          mydata[i*threadIdx.x % 8]);
}
```

Shared Memory: Use Cases

- inter-thread communication
- reduce global memory access → manual cache
- adjust global memory access pattern
- indexed access
- combine costly operations

```
__global__ void kernel(uint *global_count, ...)
{
    __shared__ uint blockcount;
    blockcount = 0;
    __syncthreads();
    uint myoffset = atomicAdd(&blockcount, myadd);
    __syncthreads();
    if(threadIdx.x == 0)
        blockcount = atomicAdd(global_count, blockcount);
    __syncthreads();
    myoffset += blockcount;
}
```

Registers vs Shared vs Global Memory



https://cuda-tutorial.github.io

References

- NVIDIA, *CUDA Programming Guide*
- NVIDIA, *NVCC Documentation*
- NVIDIA, *PTX ISA*
- NVIDIA, *CUDA Binary Utilities*

- NVIDIA, *Ampere GA102 GPU Architecture Whitepaper*, 2020

- E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, *NVIDIA Tesla: A Unified Graphics and Computing Architecture*, in IEEE Micro, vol. 28, no. 2, pp. 39–55, 2008, doi: 10.1109/MM.2008.31.

- Z. Jia, M. Maggioni, B. Staiger, D. P. Scarpazza, *Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking*, arXiv, 2018, arXiv: 1804.06826

References

- B. W. Coon, J. R. Nickolls, J. E. Lindholm, S. D. Tzvetkov, *Structured programming control flow in a SIMD architecture*, 2007, US Patent 7877585B1
- B. W. Coon, J. E. Lindholm, P. C. Mills, J. R. Nickolls, *Processing an indirect branch instruction in a SIMD architecture*, 2006, US Patent 7761697B1
- B. W. Coon, J. E. Lindholm, *Systems and method for managing divergent threads in a SIMD architecture*, 2005, US Patent 8667256B1
- J. E. Lindholm, M. C. Shebanow, *Tree-based thread management*, 2014, US Patent 9830161B2