King Abdullah University of Science and Technology KAUST

CS 380 - GPU and GPGPU Programming
Lecture 6: GPU Architecture, Pt. 4

Markus Hadwiger, KAUST

Reading Assignment #3 (until Sep 16)

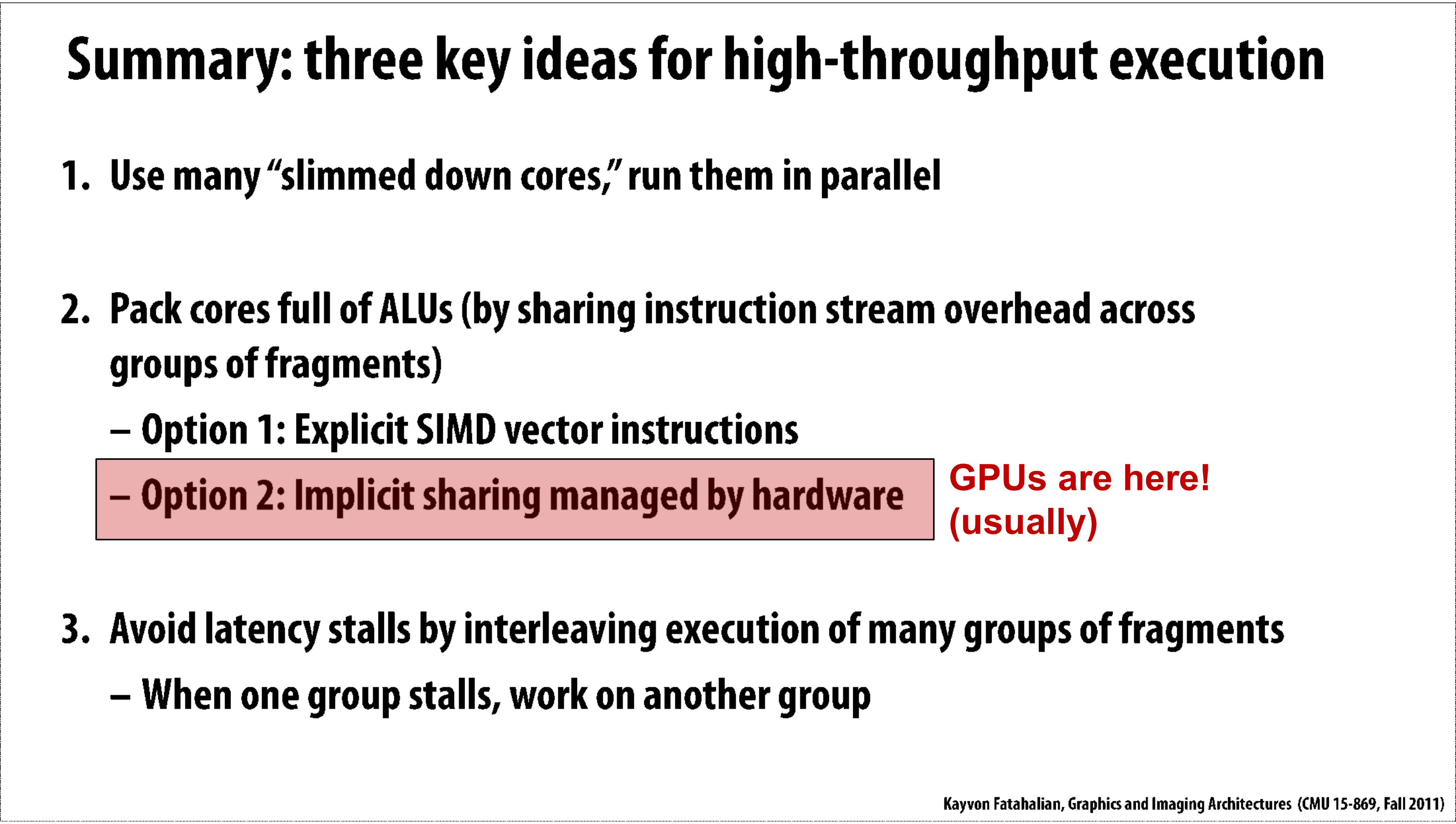
Read (required):

- Programming Massively Parallel Processors book, 4th edition,
  Chapter 4 (*Compute architecture and scheduling*)

- NVIDIA CUDA C++ Programming Guide (current: v12.6, Aug 29, 2024):
  *Read **Chapter 5.6*** (Compute Capability);
  *Read **Chapter 19.1*** (Compute Capabilities);
  *Browse all of **Chapter 19*** (Compute Capabilities)
  *Browse all of **Chapter 8.2*** (Maximize Utilization) and
  Chapter 8.4 (Maximize Instruction Throughput)

`https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`

Where this is going...



Summary: three key ideas for high-throughput execution

1. Use many "slimmed down cores," run them in parallel

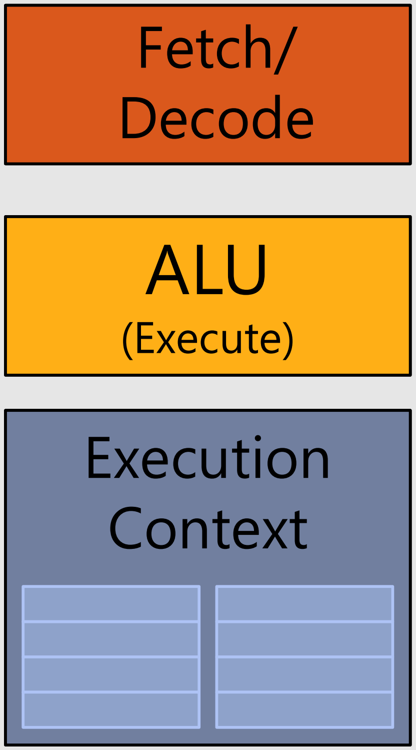
2. Pack cores full of ALUs (by sharing instruction stream overhead across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

GPUs are here!
(usually)

3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group

Kayvon Fatahalian, Graphics and Imaging Architectures (CMU 15-869, Fall 2011)

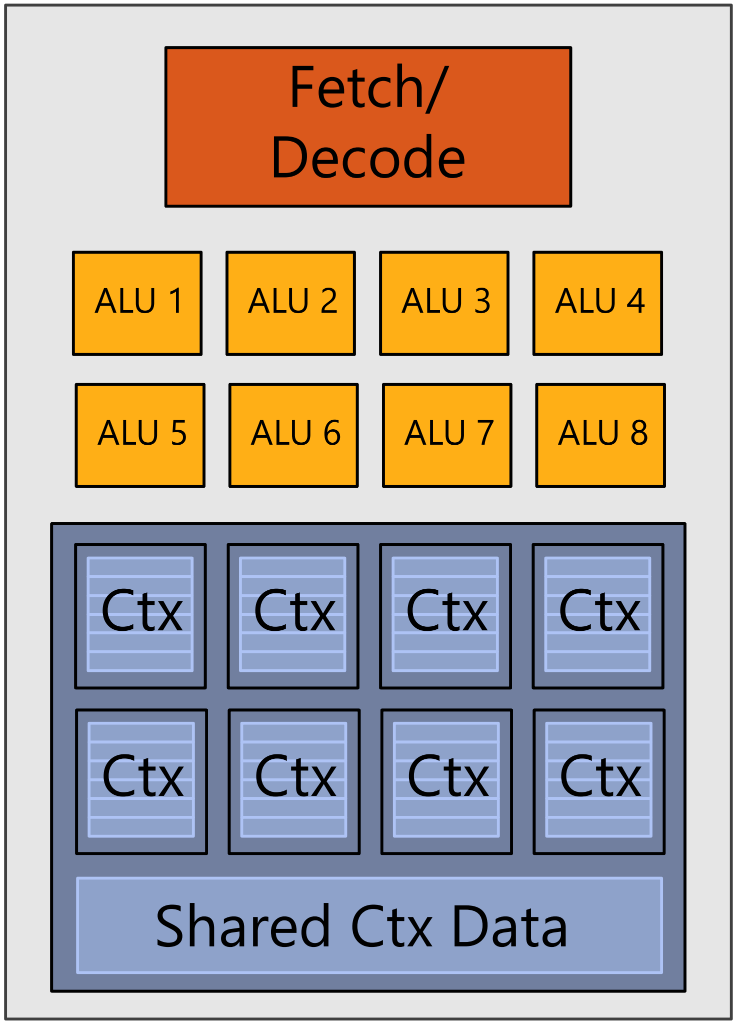
Idea #1: Slim down



Idea #1:

Remove components that help a single instruction stream run fast (i.e., that reduce latency of a single instruction stream)

Idea #2: Add ALUs (sharing inst. stream)



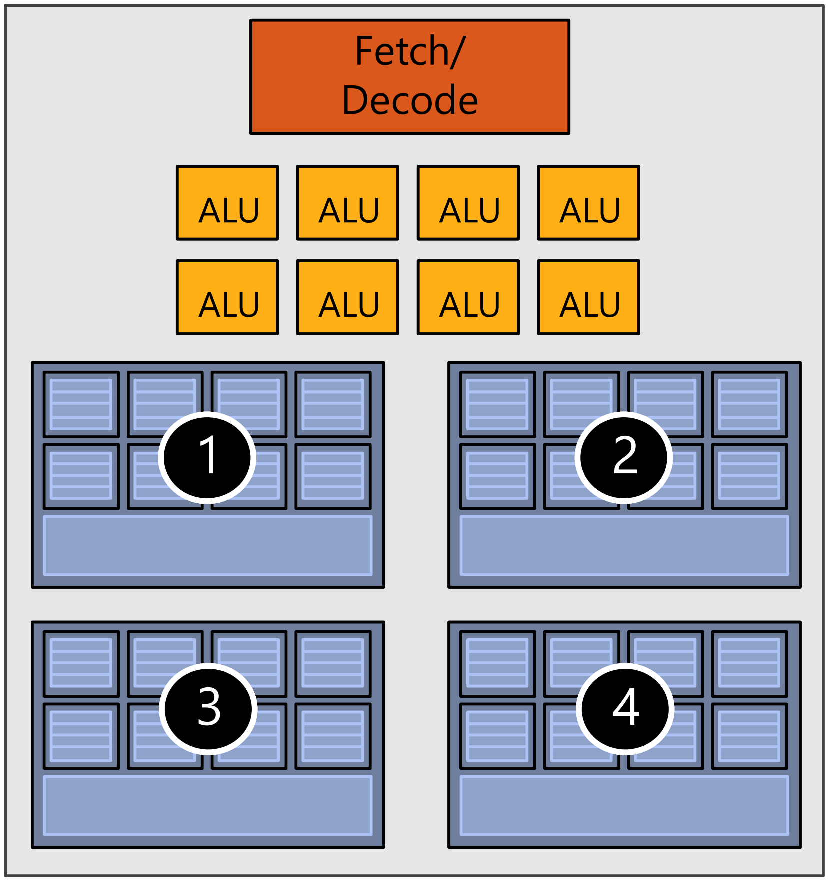
Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

(or SIMT, SPMD)

Idea #3: Store multiple group contexts



Idea #3:

Interleave execution of groups of threads

(the number of groups is *not fixed*, but depends on the context storage requirements of a given kernel!)

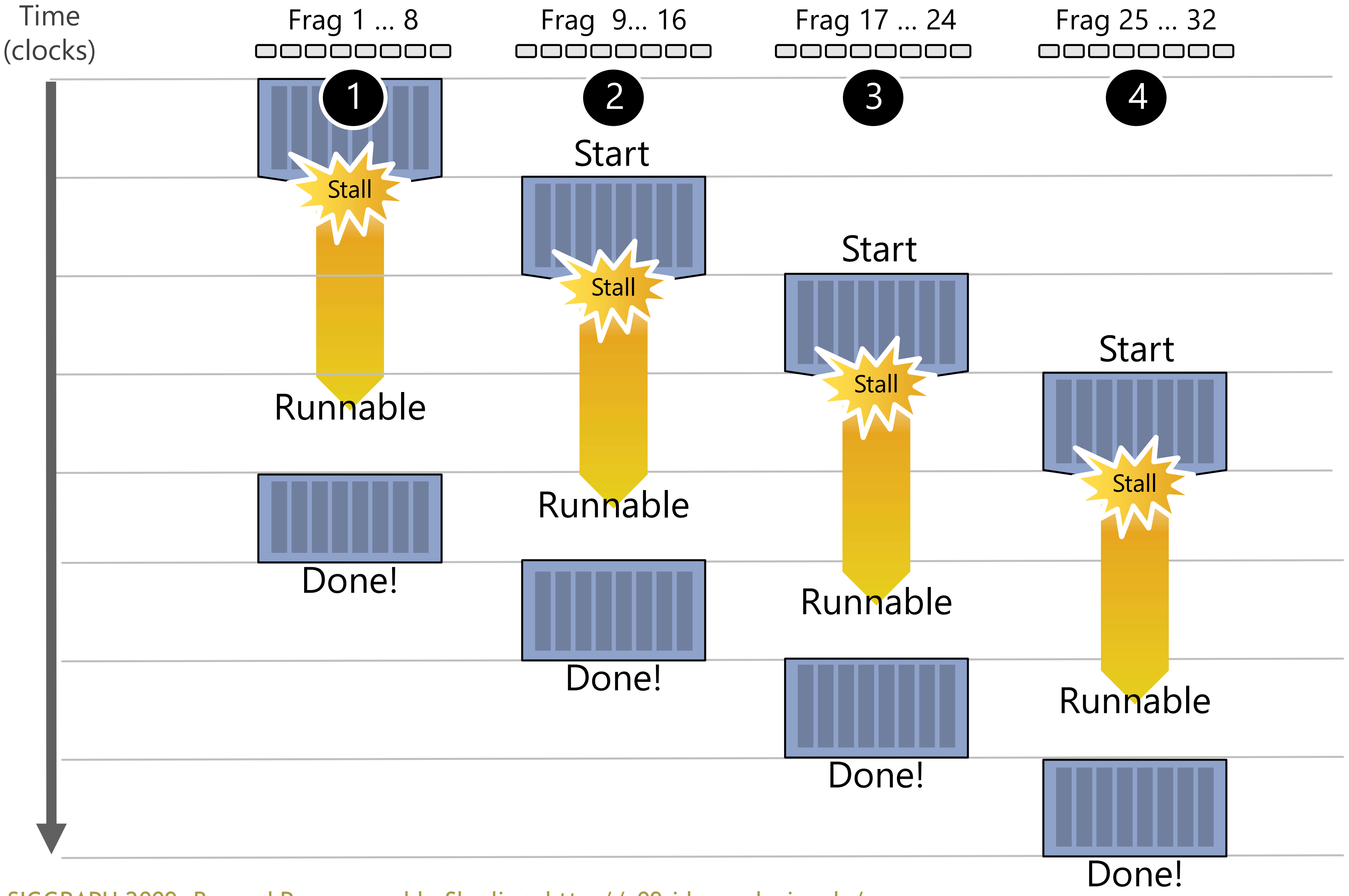
Idea #3: Interleave execution of groups

But we have LOTS of independent fragments.

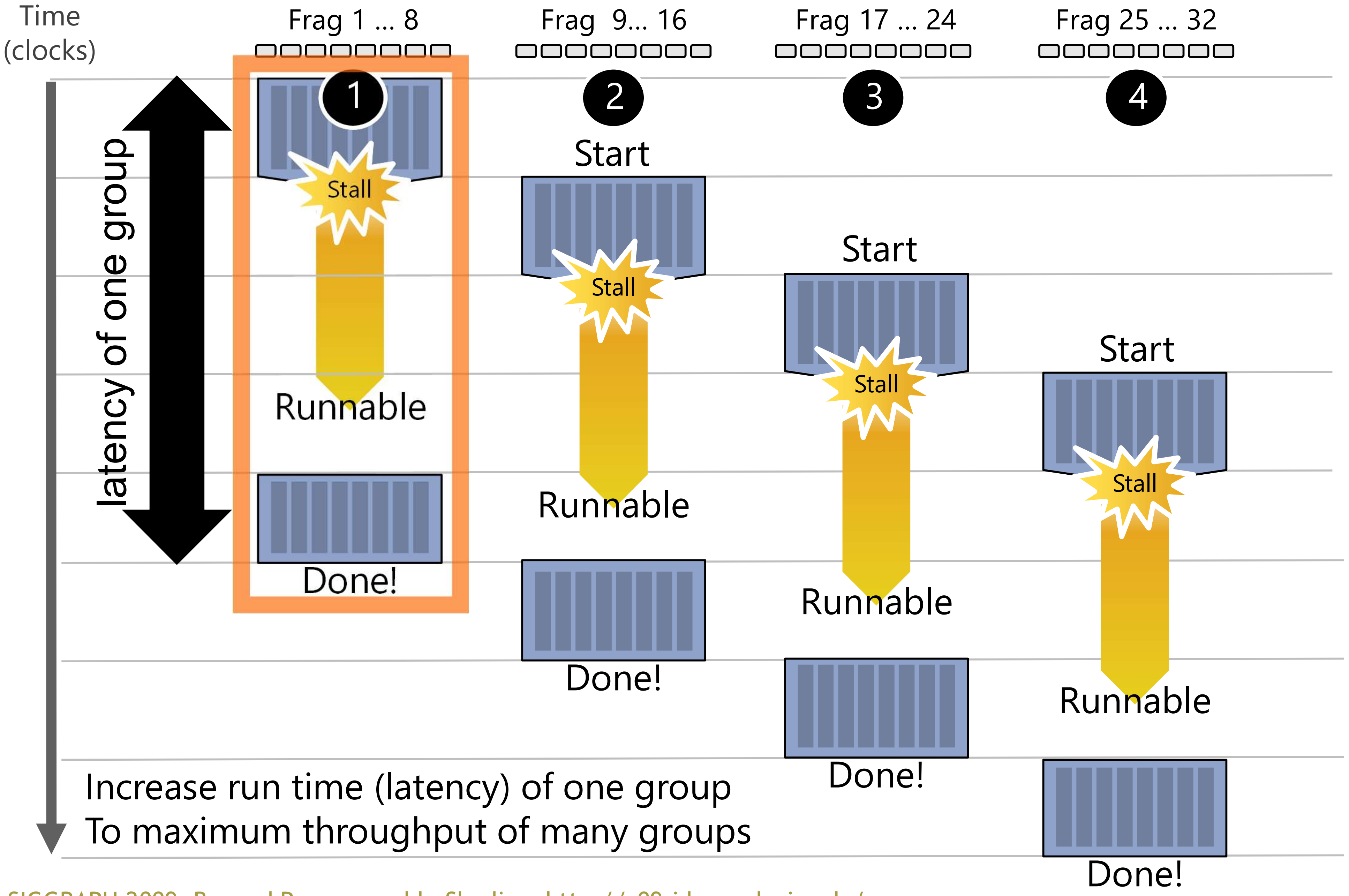
Idea #3:

Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

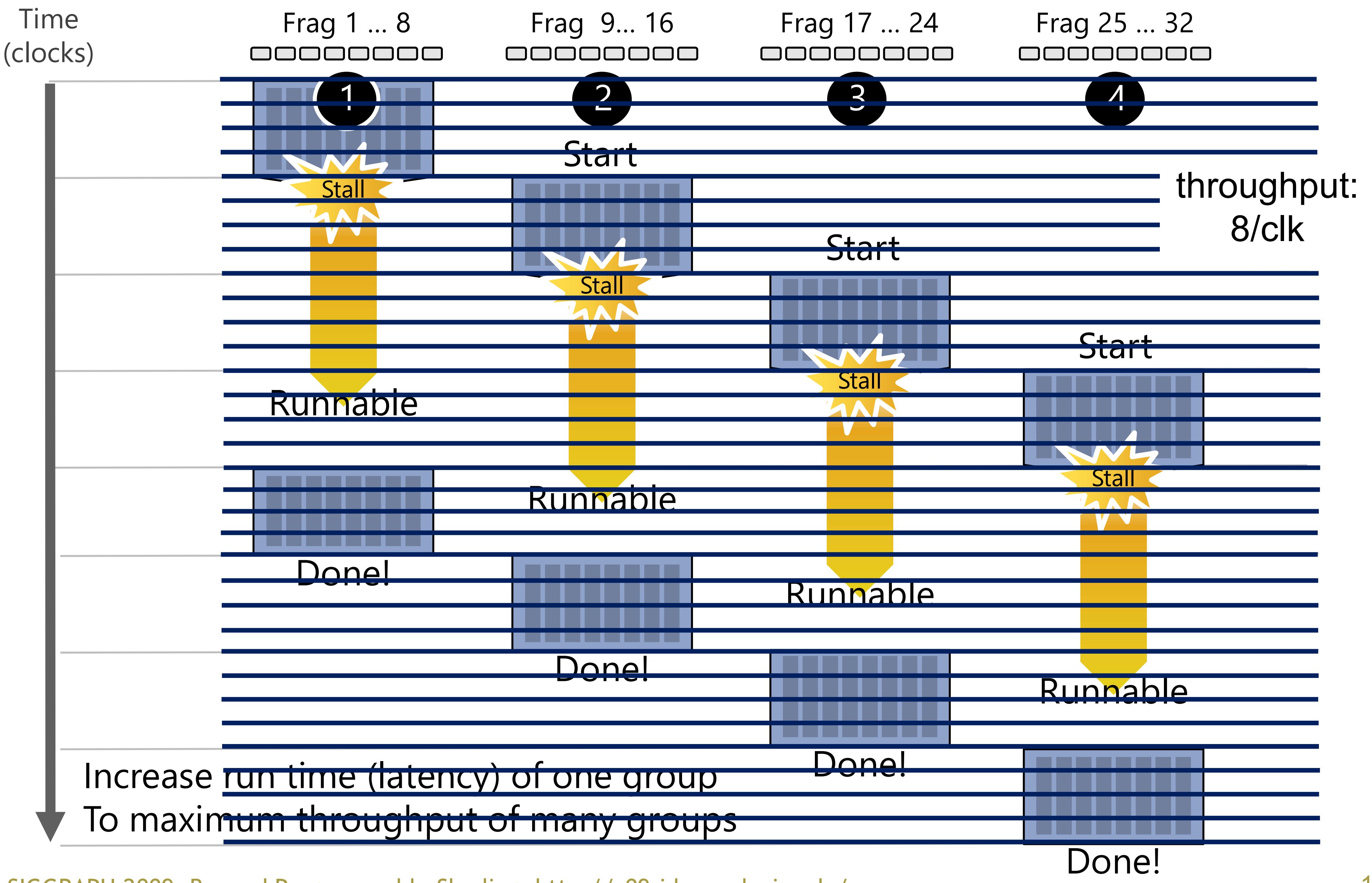


Hiding shader stalls



Increase run time (latency) of one group
To maximum throughput of many groups

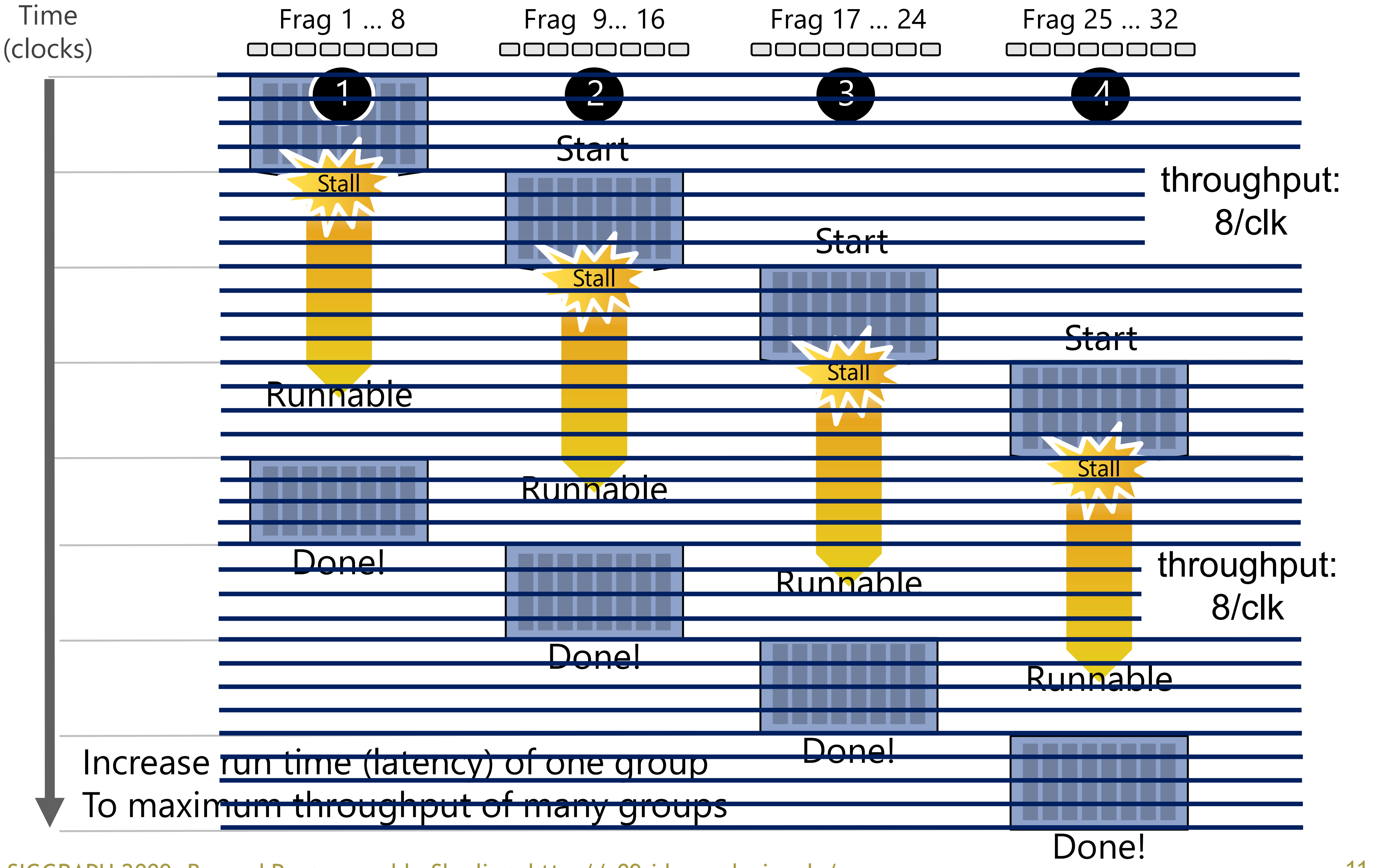
Throughput! (4 groups of threads)



throughput:
8/clk

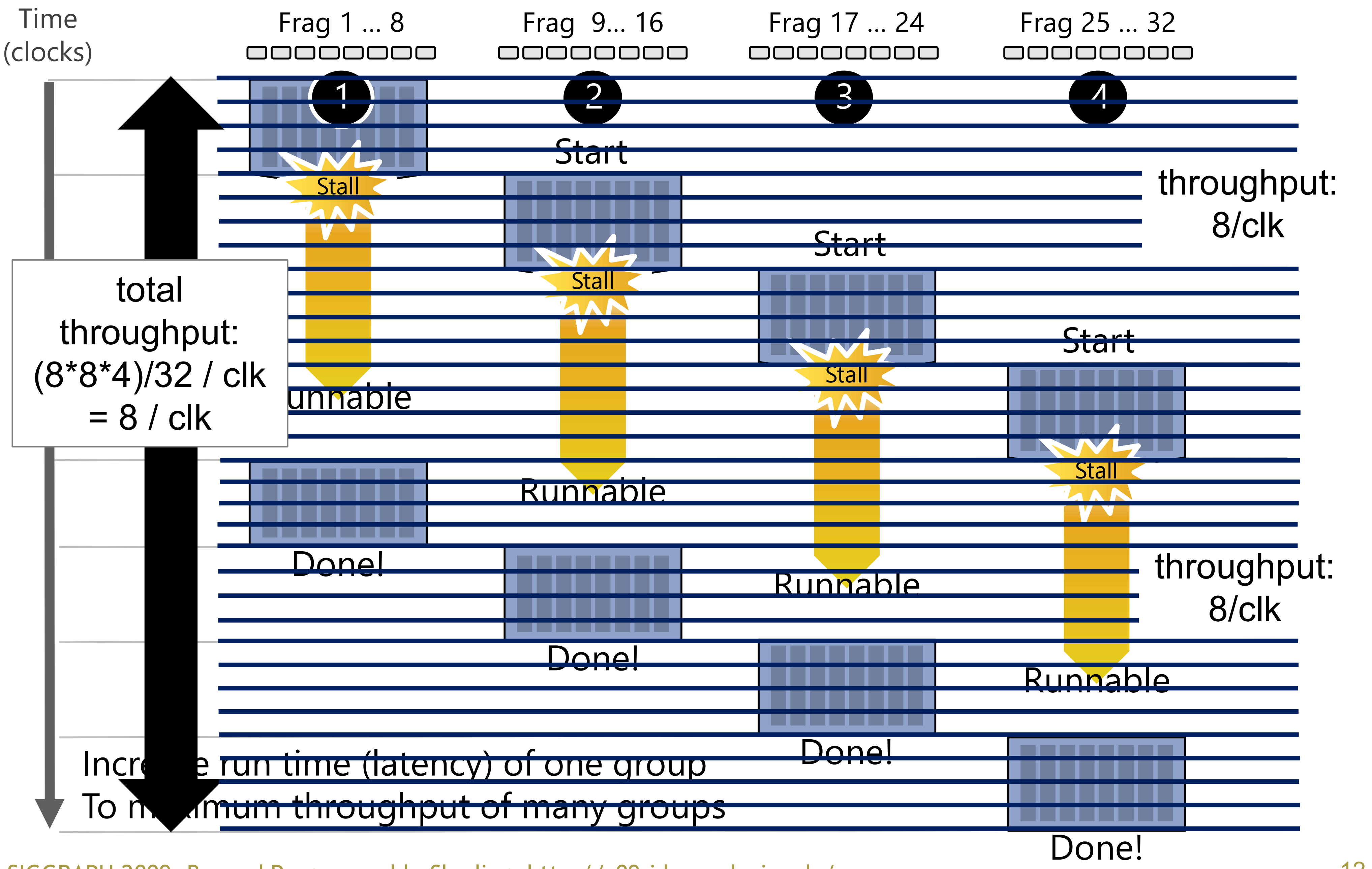
Increase run time (latency) of one group
To maximum throughput of many groups

Throughput! (4 groups of threads)



Increase run time (latency) of one group
To maximum throughput of many groups

Throughput! (4 groups of threads)



Time (clocks)

Frag 1 … 8

Frag 9… 16

Frag 17 … 24

Frag 25 … 32

1

2

3

4

Stall

Runnable

Done!

Start

Stall

Runnable

Done!

Start

Stall

Runnable

Done!

Start

Stall

Runnable

Done!

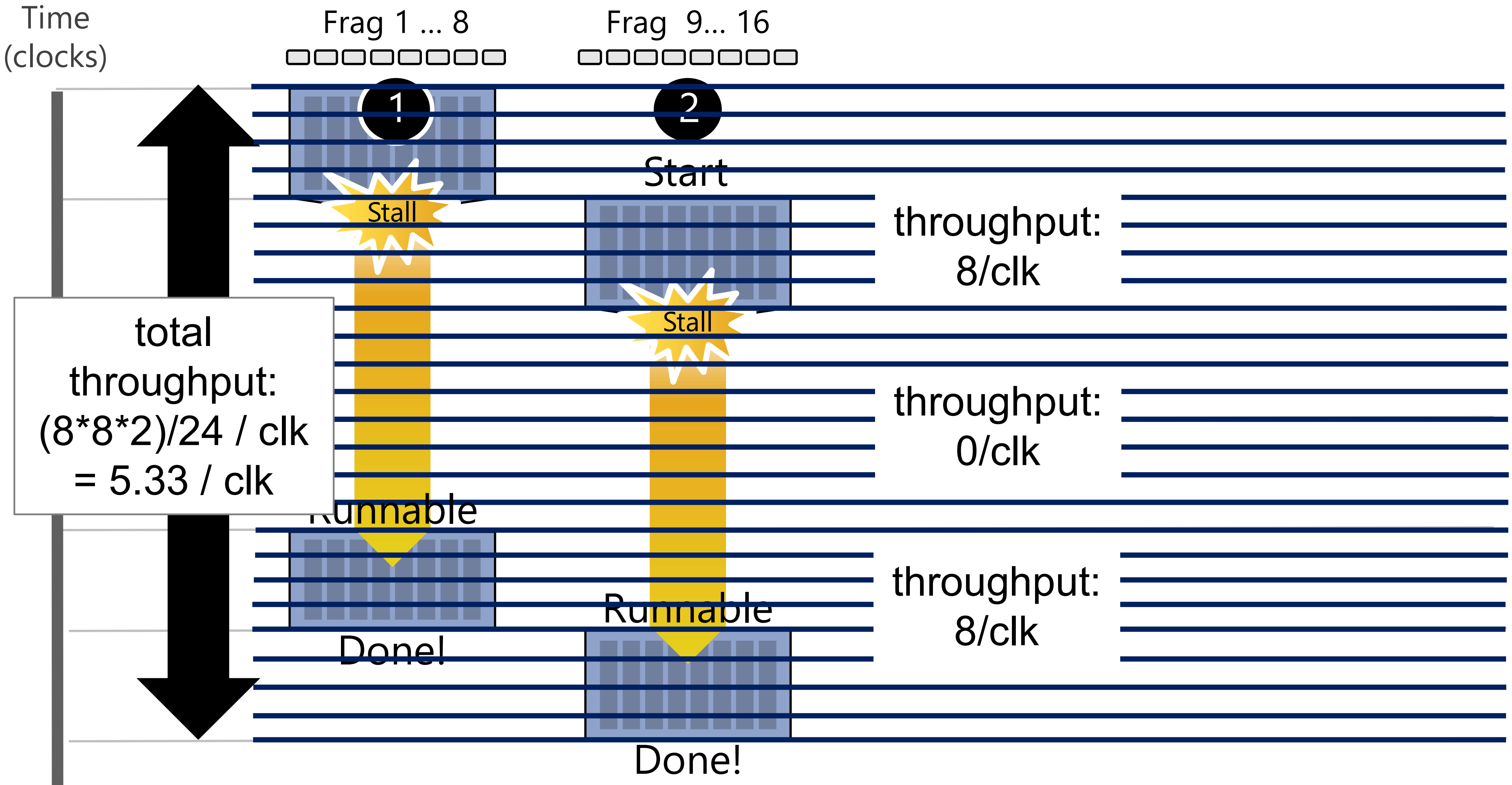
throughput: 8/clk

throughput: 8/clk

total throughput: (8*8*4)/32 / clk = 8 / clk

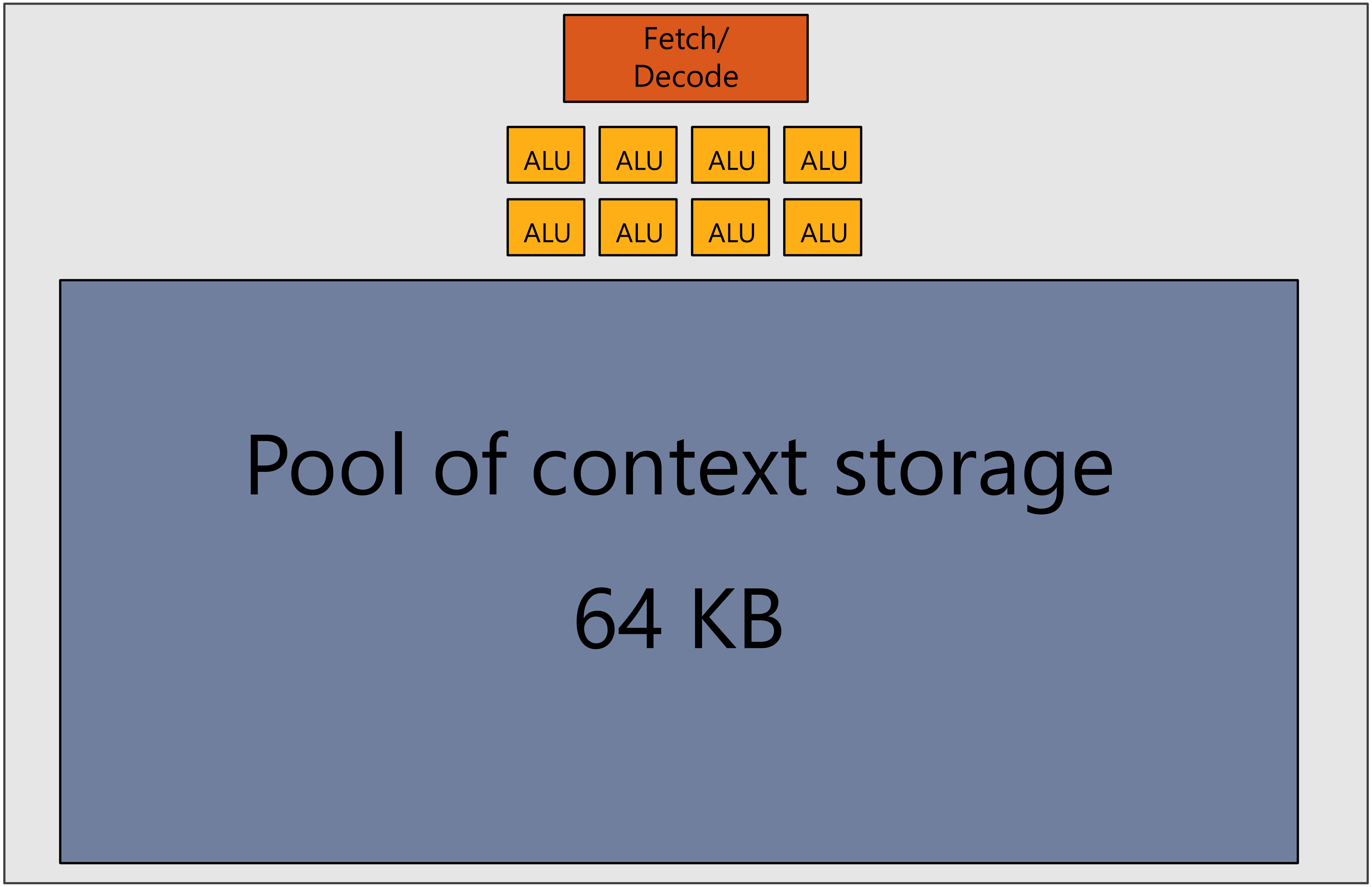
Increase run time (latency) of one group
To maximum throughput of many groups

Throughput! (2 groups of threads)



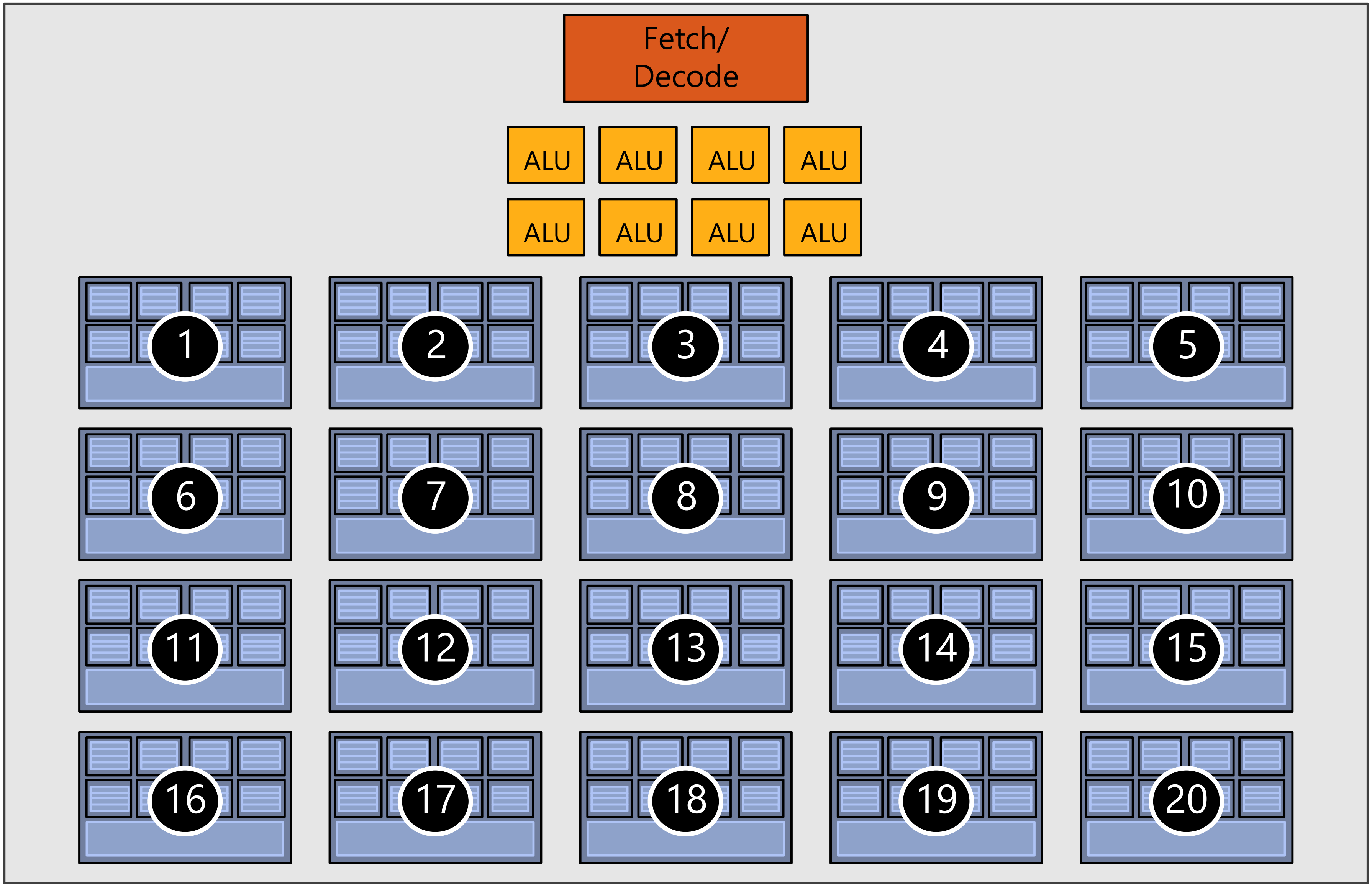
Increase run time (latency) of one group
To maximum throughput of many groups

Idea #3: Store multiple group contexts

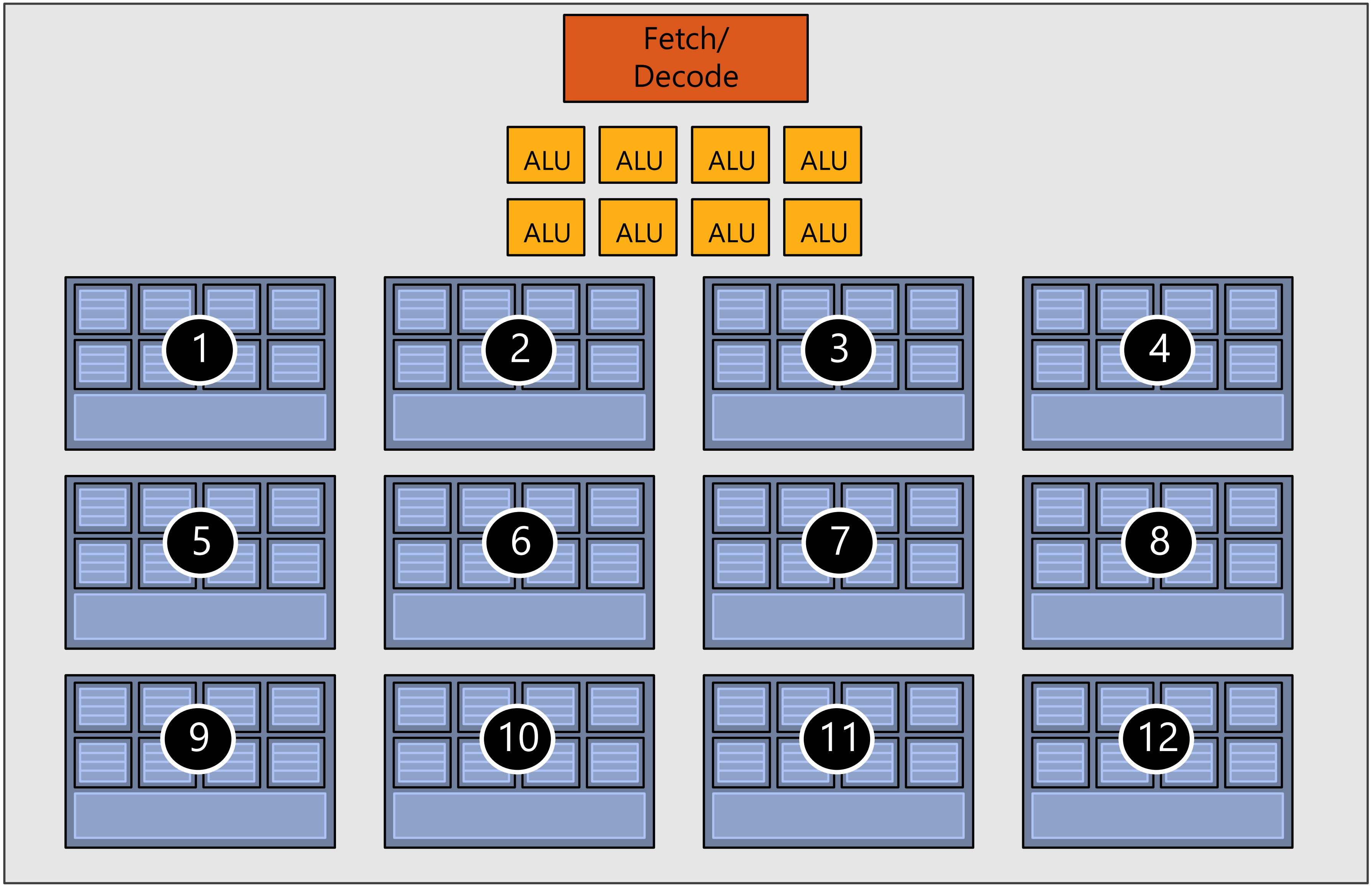


Twenty small contexts (few regs/thread)

(maximal latency hiding ability)

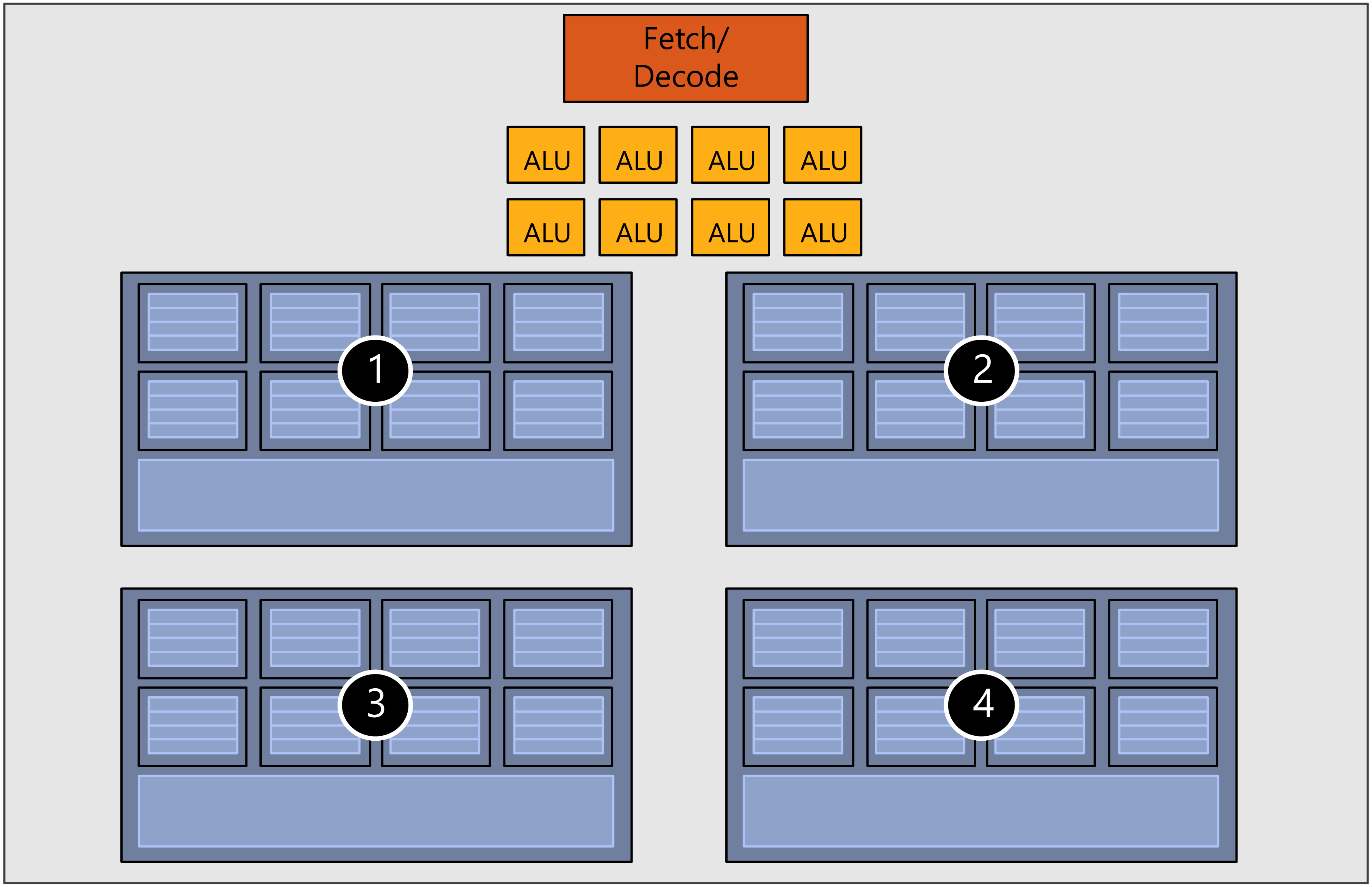


Twelve medium contexts (more regs/th.)



Four large contexts (many regs/thread)

(low latency hiding ability)



Complete GPU

16 cores

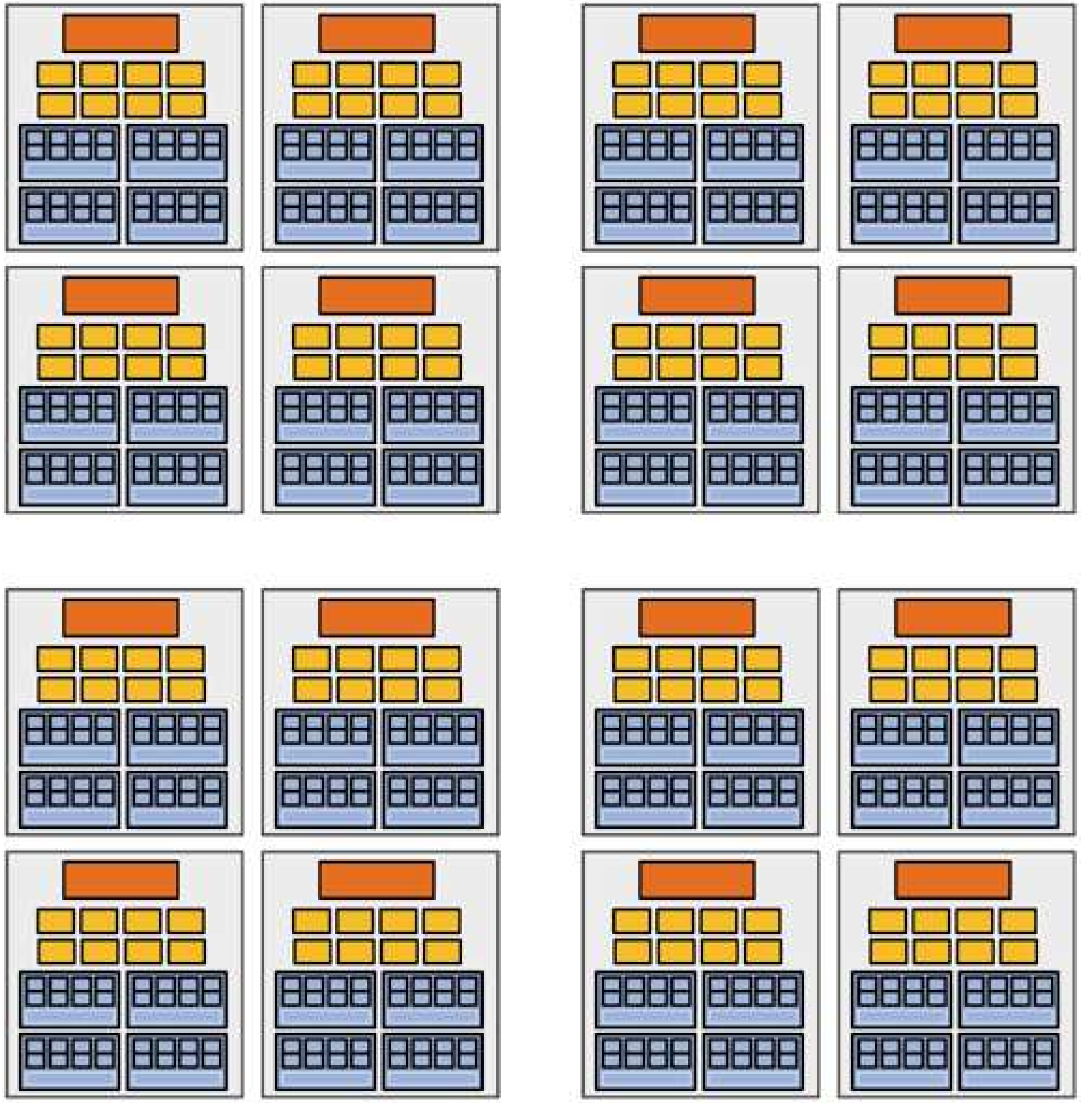
8 mul-add [mad] ALUs per core (8*16 = **128** total)

16 simultaneous instruction streams

64 (4*16) concurrent (but interleaved) instruction streams

512 (8*4*16) concurrent fragments (resident threads)

= **256 GFLOPs** (@ 1GHz)
(**128** * 2 [mad] * 1G)



Complete GPU

16 cores

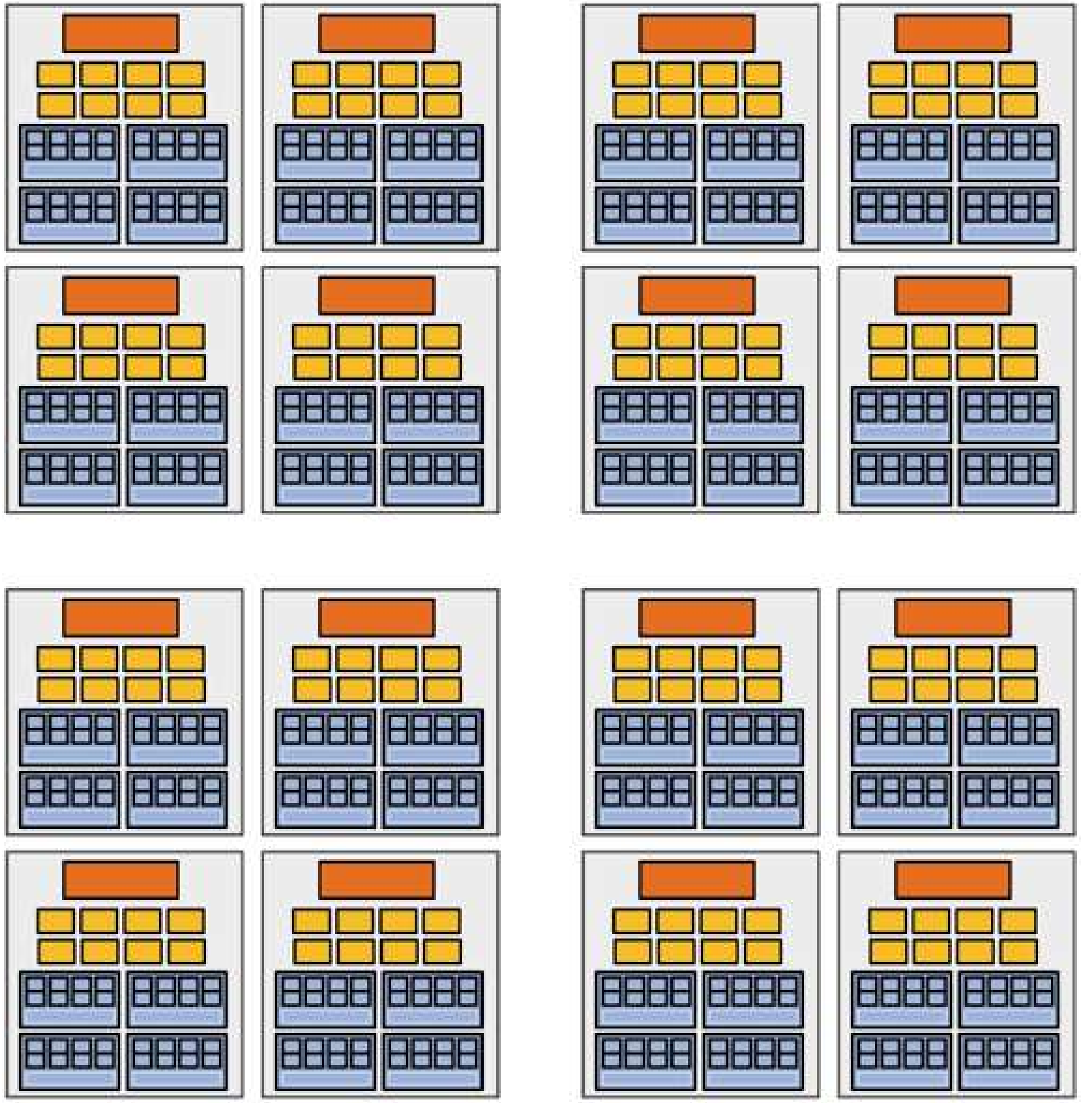
8 mul-add [mad] ALUs per core
(8*16 = **128** total)

16 simultaneous
instruction streams

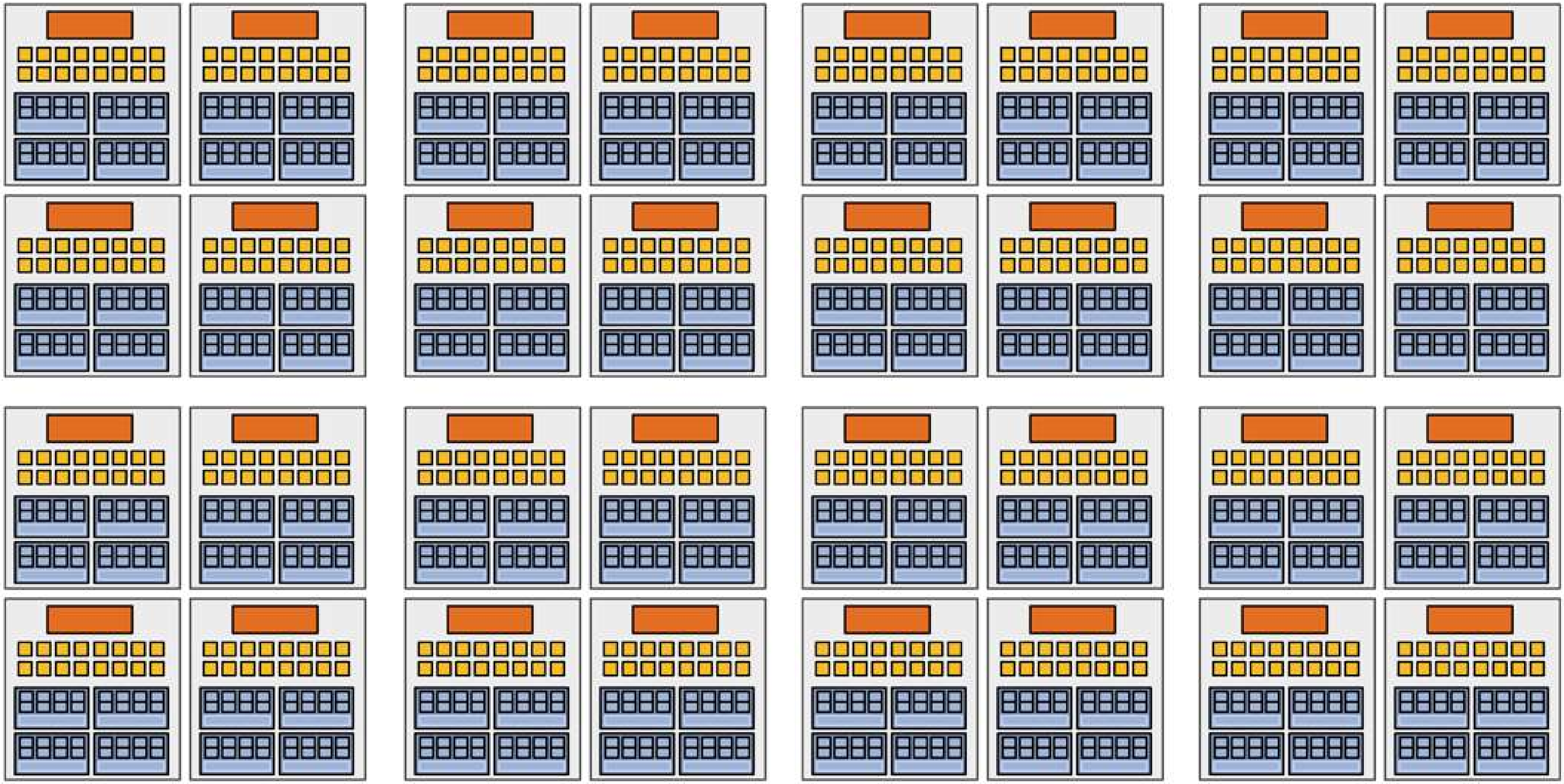
64 (4*16) concurrent (but interleaved) instruction streams

512 (8*4*16) concurrent fragments (resident threads)

= **256 GFLOPs** (@ 1GHz)
(**128** * 2 [mad] * 1G)

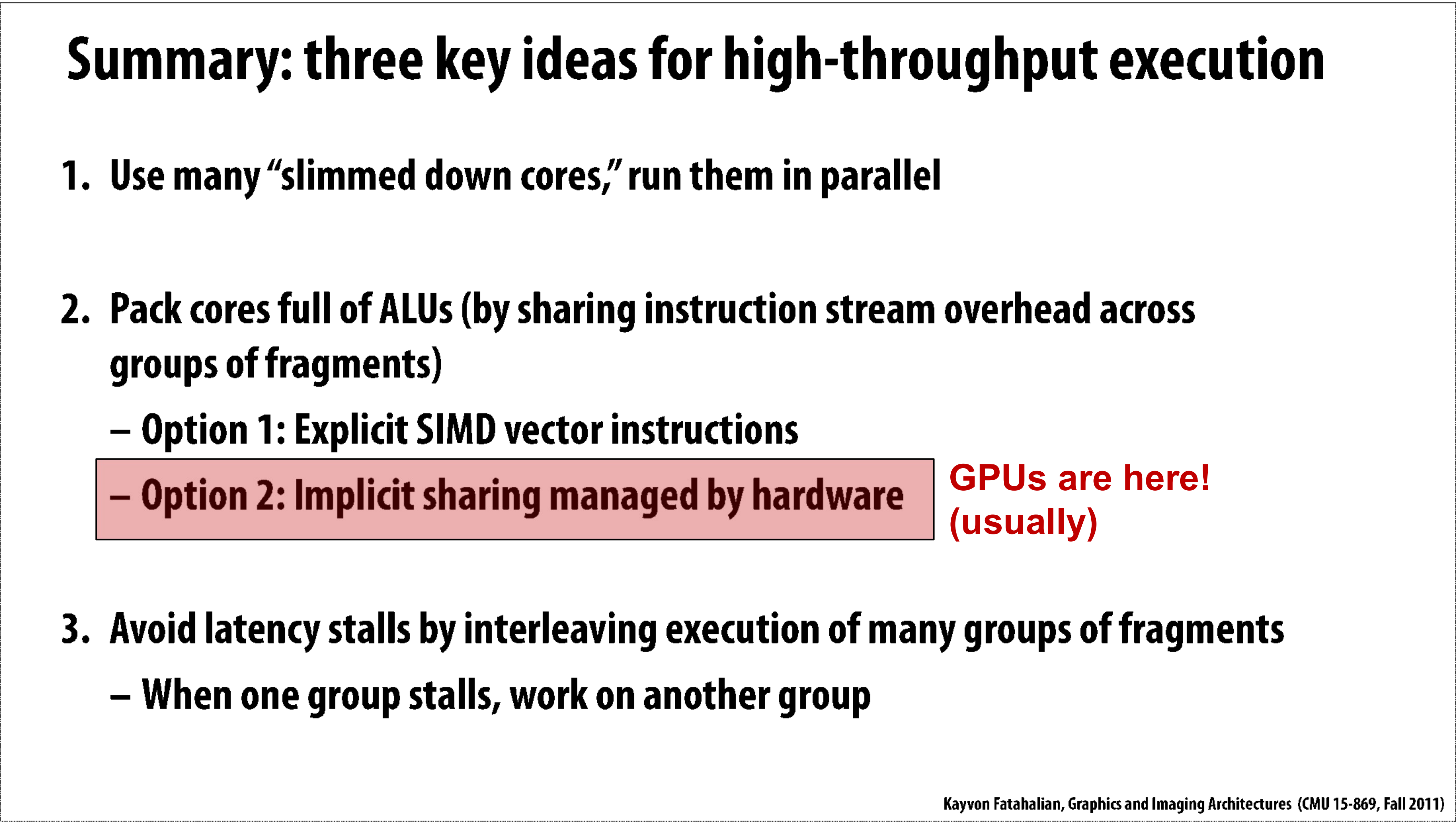


“Enthusiast” GPU (Some time ago :)



32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)

Where We‘ve Arrived...



Summary: three key ideas for high-throughput execution

1. Use many “slimmed down cores,” run them in parallel

2. Pack cores full of ALUs (by sharing instruction stream overhead across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

GPUs are here!
(usually)

3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group

Kayvon Fatahalian, Graphics and Imaging Architectures (CMU 15-869, Fall 2011)

GPU Architecture:
Real Architectures

NVIDIA Architectures (since first CUDA GPU)

Tesla [CC 1.x]: 2007-2009

- G80, G9x: 2007 (Geforce 8800, ...)
  GT200: 2008/2009 (GTX 280, ...)

Fermi [CC 2.x]: 2010 (2011, 2012, 2013, …)

- GF100, ... (GTX 480, ...)
  GF104, ... (GTX 460, ...)
  GF110, ... (GTX 580, ...)

Kepler [CC 3.x]: 2012 (2013, 2014, 2016, …)

- GK104, ... (GTX 680, ...)
  GK110, ... (GTX 780, GTX Titan, ...)

Maxwell [CC 5.x]: 2015

- GM107, ... (GTX 750Ti, ...)
  GM204, ... (GTX 980, Titan X, ...)

Pascal [CC 6.x]: 2016 (2017, 2018, 2021, 2022, …)

- GP100 (Tesla P100, ...)
- GP10x: x=2,4,6,7,8, ...
  (GTX 1060, 1070, 1080, Titan X *Pascal*, Titan Xp, ...)

Volta [CC 7.0, 7.2]: 2017/2018

- GV100, ...
  (Tesla V100, Titan V, Quadro GV100, ...)

Turing [CC 7.5]: 2018/2019

- TU102, TU104, TU106, TU116, TU117, ...
  (Titan RTX, RTX 2070, 2080 (Ti), GTX 1650, 1660, ...)

Ampere [CC 8.0, 8.6, 8.7]: 2020

- GA100, GA102, GA104, GA106, ...
  (A100, RTX 3070, 3080, 3090 (Ti), RTX A6000, ...)

Hopper [CC 9.0], Ada Lovelace [CC 8.9]: 2022/23

- GH100, AD102, AD103, AD104, ...
  (H100, L40, RTX 4080 (12/16 GB), 4090, RTX 6000, ...)

Blackwell [CC 10.0]: *coming in 2024/25*

- GB200/GB202, GB20x, ...?
  (RTX 5080/5090, GB200 NVL72, HGX B100/200, ...?)

see `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`
and `https://en.wikipedia.org/wiki/CUDA`

Interlude: PTX vs. SASS Code (1)

PTX is virtual machine ISA

SASS is actual machine ISA

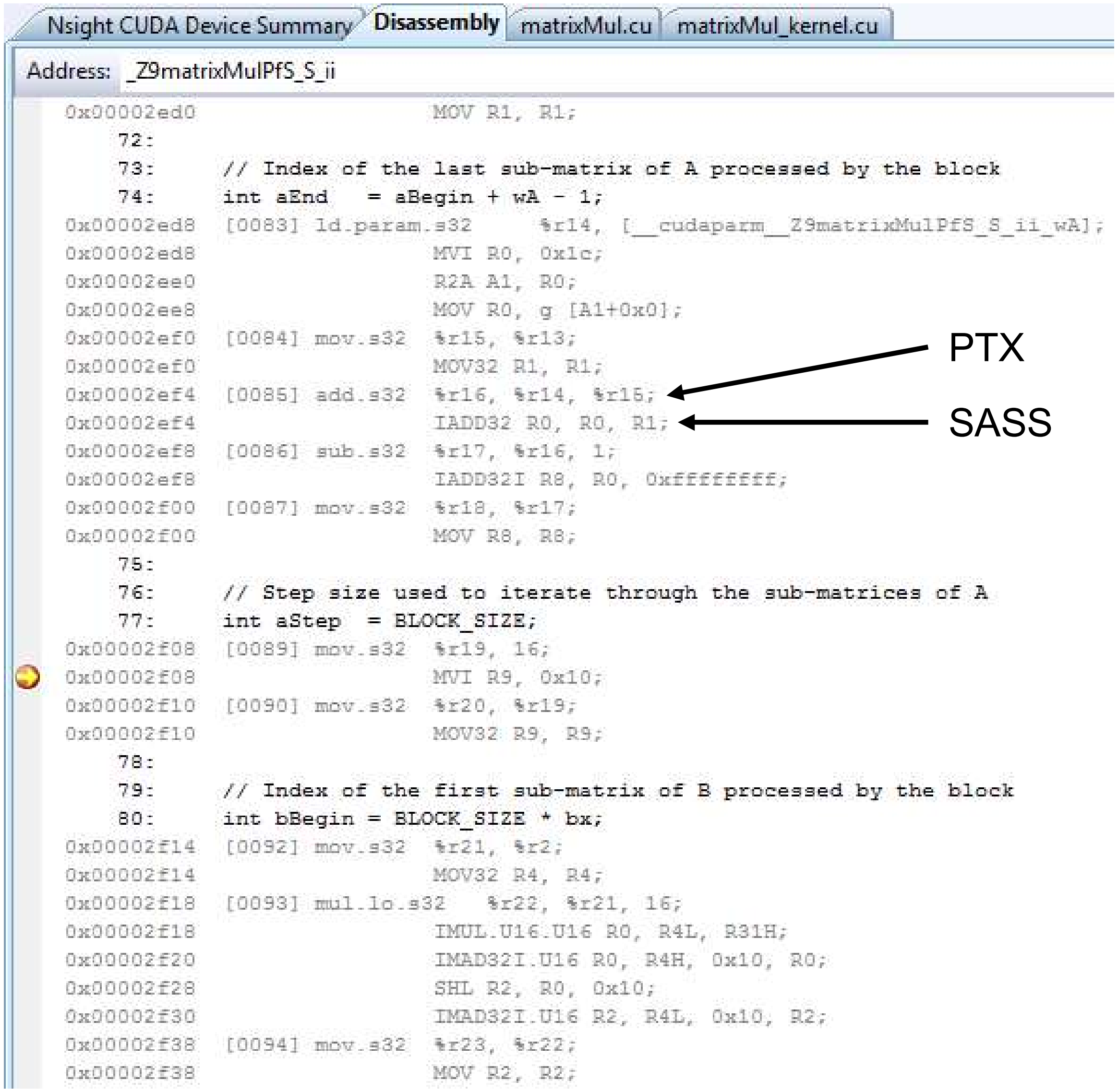
For disassembly:

`cuobjdump` / `nvdisasm`

See `CUDA_Binary_Utilities.pdf`

For debugging (and code inspection) see:

`https://developer.nvidia.com/nsight-visual-studio-edition`



Address: _Z9matrixMulPfS_S_ii

```
0x00002ed0                    MOV R1, R1;
     72:
     73:     // Index of the last sub-matrix of A processed by the block
     74:     int aEnd    = aBegin + wA - 1;
0x00002ed8  [0083] ld.param.s32     %r14, [__cudaparm__Z9matrixMulPfS_S_ii_wA];
0x00002ed8                    MVI R0, 0x1c;
0x00002ee0                    R2A A1, R0;
0x00002ee8                    MOV R0, g [A1+0x0];
0x00002ef0  [0084] mov.s32   %r15, %r13;
0x00002ef0                    MOV32 R1, R1;
0x00002ef4  [0085] add.s32   %r16, %r14, %r15;
0x00002ef4                    IADD32 R0, R0, R1;
0x00002ef8  [0086] sub.s32   %r17, %r16, 1;
0x00002ef8                    IADD32I R8, R0, 0xffffffff;
0x00002f00  [0087] mov.s32   %r18, %r17;
0x00002f00                    MOV R8, R8;
     75:
     76:     // Step size used to iterate through the sub-matrices of A
     77:     int aStep   = BLOCK_SIZE;
0x00002f08  [0089] mov.s32   %r19, 16;
0x00002f08                    MVI R9, 0x10;
0x00002f10  [0090] mov.s32   %r20, %r19;
0x00002f10                    MOV32 R9, R9;
     78:
     79:     // Index of the first sub-matrix of B processed by the block
     80:     int bBegin = BLOCK_SIZE * bx;
0x00002f14  [0092] mov.s32   %r21, %r2;
0x00002f14                    MOV32 R4, R4;
0x00002f18  [0093] mul.lo.s32    %r22, %r21, 16;
0x00002f18                    IMUL.U16.U16 R0, R4L, R31H;
0x00002f20                    IMAD32I.U16 R0, R4H, 0x10, R0;
0x00002f28                    SHL R2, R0, 0x10;
0x00002f30                    IMAD32I.U16 R2, R4L, 0x10, R2;
0x00002f38  [0094] mov.s32   %r23, %r22;
0x00002f38                    MOV R2, R2;
```

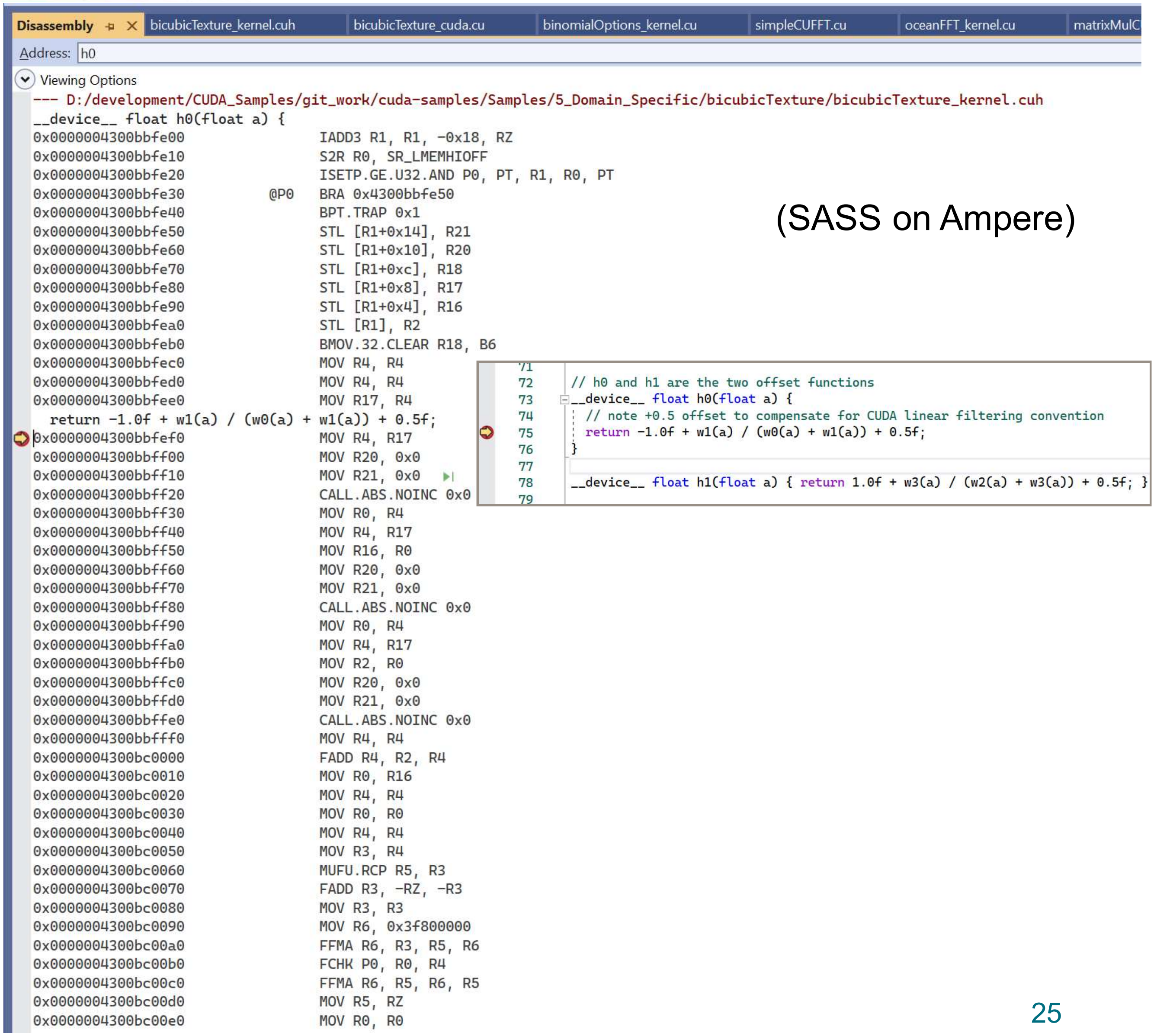
PTX

SASS

Interlude: PTX vs. SASS Code (2)

Note

- Size of instructions (here: 16 bytes)
- **`MUFU.RCP`** computing FP32 reciprocal on SFU (there is no SASS division: division is an algorithm comprising simpler instructions)
- This is debug code: redundant register moves not (yet) removed by optimizer in assembler *(result of virtual PTX registers being mapped to same physical register)*
- …



(SASS on Ampere)

```
--- D:/development/CUDA_Samples/git_work/cuda-samples/Samples/5_Domain_Specific/bicubicTexture/bicubicTexture_kernel.cuh
__device__ float h0(float a) {
0x0000004300bbfe00          IADD3 R1, R1, -0x18, RZ
0x0000004300bbfe10          S2R R0, SR_LMEMHIOFF
0x0000004300bbfe20          ISETP.GE.U32.AND P0, PT, R1, R0, PT
0x0000004300bbfe30      @P0 BRA 0x4300bbfe50
0x0000004300bbfe40          BPT.TRAP 0x1
0x0000004300bbfe50          STL [R1+0x14], R21
0x0000004300bbfe60          STL [R1+0x10], R20
0x0000004300bbfe70          STL [R1+0xc], R18
0x0000004300bbfe80          STL [R1+0x8], R17
0x0000004300bbfe90          STL [R1+0x4], R16
0x0000004300bbfea0          STL [R1], R2
0x0000004300bbfeb0          BMOV.32.CLEAR R18, B6
0x0000004300bbfec0          MOV R4, R4
0x0000004300bbfed0          MOV R4, R4
0x0000004300bbfee0          MOV R17, R4
  return -1.0f + w1(a) / (w0(a) + w1(a)) + 0.5f;
0x0000004300bbfef0          MOV R4, R17
0x0000004300bbff00          MOV R20, 0x0
0x0000004300bbff10          MOV R21, 0x0
0x0000004300bbff20          CALL.ABS.NOINC 0x0
0x0000004300bbff30          MOV R0, R4
0x0000004300bbff40          MOV R4, R17
0x0000004300bbff50          MOV R16, R0
0x0000004300bbff60          MOV R20, 0x0
0x0000004300bbff70          MOV R21, 0x0
0x0000004300bbff80          CALL.ABS.NOINC 0x0
0x0000004300bbff90          MOV R0, R4
0x0000004300bbffa0          MOV R4, R17
0x0000004300bbffb0          MOV R2, R0
0x0000004300bbffc0          MOV R20, 0x0
0x0000004300bbffd0          MOV R21, 0x0
0x0000004300bbffe0          CALL.ABS.NOINC 0x0
0x0000004300bbfff0          MOV R4, R4
0x0000004300bc0000          FADD R4, R2, R4
0x0000004300bc0010          MOV R0, R16
0x0000004300bc0020          MOV R4, R4
0x0000004300bc0030          MOV R0, R0
0x0000004300bc0040          MOV R4, R4
0x0000004300bc0050          MOV R3, R4
0x0000004300bc0060          MUFU.RCP R5, R3
0x0000004300bc0070          FADD R3, -RZ, -R3
0x0000004300bc0080          MOV R3, R3
0x0000004300bc0090          MOV R6, 0x3f800000
0x0000004300bc00a0          FFMA R6, R3, R5, R6
0x0000004300bc00b0          FCHK P0, R0, R4
0x0000004300bc00c0          FFMA R6, R5, R6, R5
0x0000004300bc00d0          MOV R5, RZ
0x0000004300bc00e0          MOV R0, R0
```

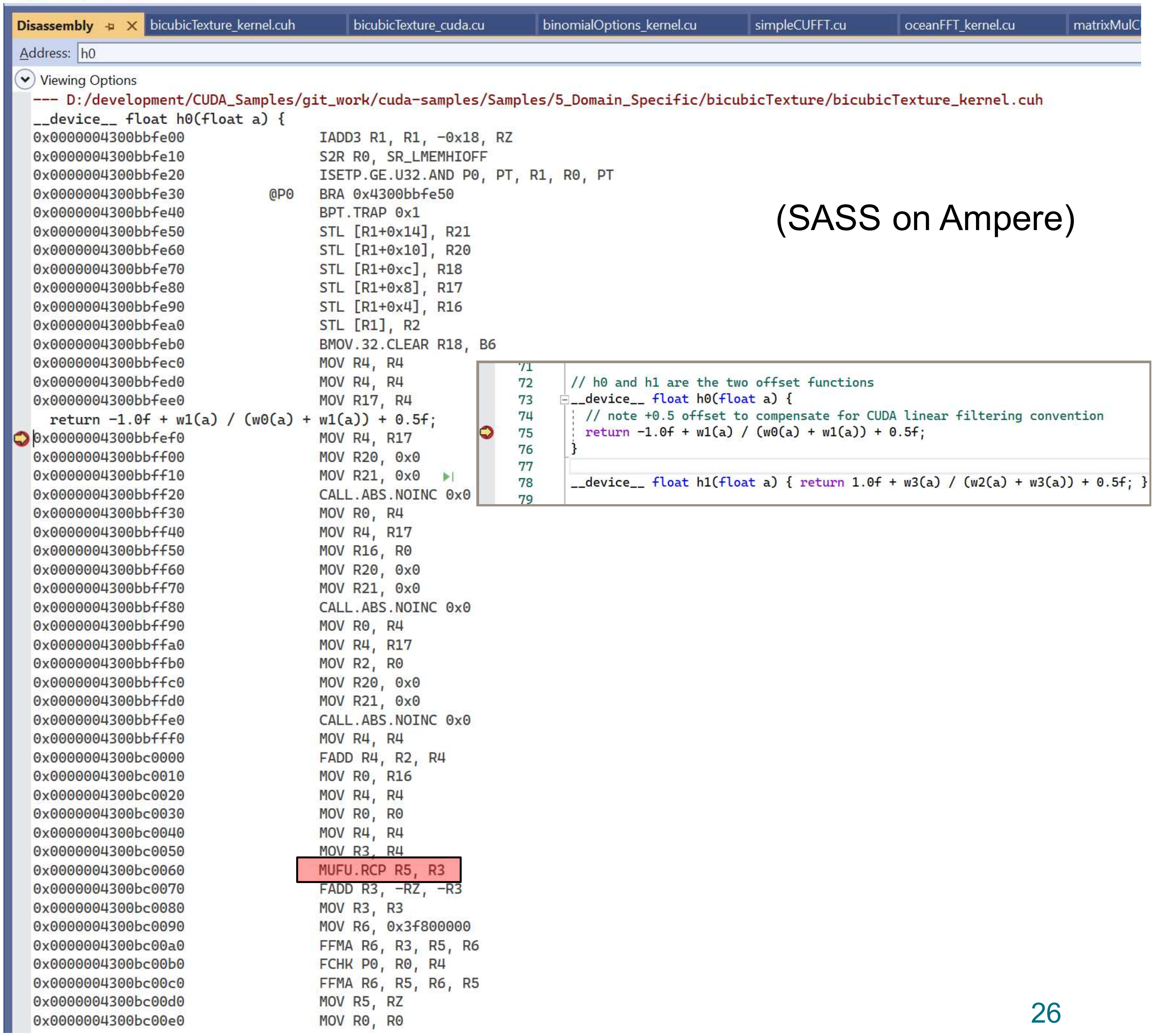
```
// h0 and h1 are the two offset functions
__device__ float h0(float a) {
  // note +0.5 offset to compensate for CUDA linear filtering convention
  return -1.0f + w1(a) / (w0(a) + w1(a)) + 0.5f;
}

__device__ float h1(float a) { return 1.0f + w3(a) / (w2(a) + w3(a)) + 0.5f; }
```

Interlude: PTX vs. SASS Code (2)

Note

- Size of instructions (here: 16 bytes)
- **`MUFU.RCP`** computing FP32 reciprocal on SFU (there is no SASS division: division is an algorithm comprising simpler instructions)
- This is debug code: redundant register moves not (yet) removed by optimizer in assembler *(result of virtual PTX registers being mapped to same physical register)*
- …



Address: h0

```
--- D:/development/CUDA_Samples/git_work/cuda-samples/Samples/5_Domain_Specific/bicubicTexture/bicubicTexture_kernel.cuh
__device__ float h0(float a) {
0x0000004300bbfe00                IADD3 R1, R1, -0x18, RZ
0x0000004300bbfe10                S2R R0, SR_LMEMHIOFF
0x0000004300bbfe20                ISETP.GE.U32.AND P0, PT, R1, R0, PT
0x0000004300bbfe30          @P0   BRA 0x4300bbfe50
0x0000004300bbfe40                BPT.TRAP 0x1
0x0000004300bbfe50                STL [R1+0x14], R21
0x0000004300bbfe60                STL [R1+0x10], R20
0x0000004300bbfe70                STL [R1+0xc], R18
0x0000004300bbfe80                STL [R1+0x8], R17
0x0000004300bbfe90                STL [R1+0x4], R16
0x0000004300bbfea0                STL [R1], R2
0x0000004300bbfeb0                BMOV.32.CLEAR R18, B6
0x0000004300bbfec0                MOV R4, R4
0x0000004300bbfed0                MOV R4, R4
0x0000004300bbfee0                MOV R17, R4
  return -1.0f + w1(a) / (w0(a) + w1(a)) + 0.5f;
0x0000004300bbfef0                MOV R4, R17
0x0000004300bbff00                MOV R20, 0x0
0x0000004300bbff10                MOV R21, 0x0
0x0000004300bbff20                CALL.ABS.NOINC 0x0
0x0000004300bbff30                MOV R0, R4
0x0000004300bbff40                MOV R4, R17
0x0000004300bbff50                MOV R16, R0
0x0000004300bbff60                MOV R20, 0x0
0x0000004300bbff70                MOV R21, 0x0
0x0000004300bbff80                CALL.ABS.NOINC 0x0
0x0000004300bbff90                MOV R0, R4
0x0000004300bbffa0                MOV R4, R17
0x0000004300bbffb0                MOV R2, R0
0x0000004300bbffc0                MOV R20, 0x0
0x0000004300bbffd0                MOV R21, 0x0
0x0000004300bbffe0                CALL.ABS.NOINC 0x0
0x0000004300bbfff0                MOV R4, R4
0x0000004300bc0000                FADD R4, R2, R4
0x0000004300bc0010                MOV R0, R16
0x0000004300bc0020                MOV R4, R4
0x0000004300bc0030                MOV R0, R0
0x0000004300bc0040                MOV R4, R4
0x0000004300bc0050                MOV R3, R4
0x0000004300bc0060                MUFU.RCP R5, R3
0x0000004300bc0070                FADD R3, -RZ, -R3
0x0000004300bc0080                MOV R3, R3
0x0000004300bc0090                MOV R6, 0x3f800000
0x0000004300bc00a0                FFMA R6, R3, R5, R6
0x0000004300bc00b0                FCHK P0, R0, R4
0x0000004300bc00c0                FFMA R6, R5, R6, R5
0x0000004300bc00d0                MOV R5, RZ
0x0000004300bc00e0                MOV R0, R0
```

(SASS on Ampere)

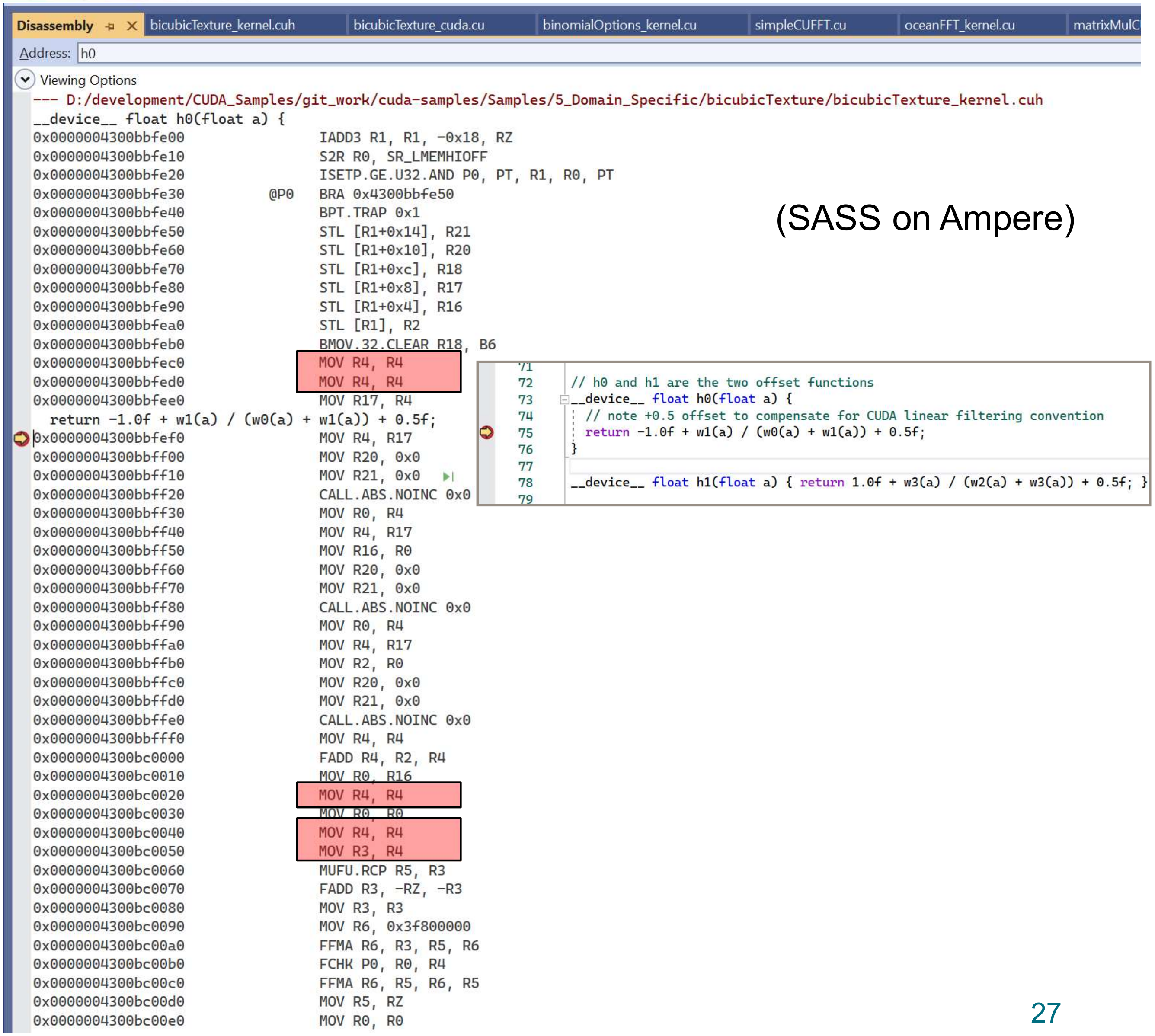
```
// h0 and h1 are the two offset functions
__device__ float h0(float a) {
  // note +0.5 offset to compensate for CUDA linear filtering convention
  return -1.0f + w1(a) / (w0(a) + w1(a)) + 0.5f;
}

__device__ float h1(float a) { return 1.0f + w3(a) / (w2(a) + w3(a)) + 0.5f; }
```

Interlude: PTX vs. SASS Code (2)

Note

- Size of instructions (here: 16 bytes)
- **`MUFU.RCP`** computing FP32 reciprocal on SFU (there is no SASS division: division is an algorithm comprising simpler instructions)
- This is debug code: redundant register moves not (yet) removed by optimizer in assembler *(result of virtual PTX registers being mapped to same physical register)*
- …



(SASS on Ampere)

```
--- D:/development/CUDA_Samples/git_work/cuda-samples/Samples/5_Domain_Specific/bicubicTexture/bicubicTexture_kernel.cuh
__device__ float h0(float a) {
0x0000004300bbfe00          IADD3 R1, R1, -0x18, RZ
0x0000004300bbfe10          S2R R0, SR_LMEMHIOFF
0x0000004300bbfe20          ISETP.GE.U32.AND P0, PT, R1, R0, PT
0x0000004300bbfe30     @P0  BRA 0x4300bbfe50
0x0000004300bbfe40          BPT.TRAP 0x1
0x0000004300bbfe50          STL [R1+0x14], R21
0x0000004300bbfe60          STL [R1+0x10], R20
0x0000004300bbfe70          STL [R1+0xc], R18
0x0000004300bbfe80          STL [R1+0x8], R17
0x0000004300bbfe90          STL [R1+0x4], R16
0x0000004300bbfea0          STL [R1], R2
0x0000004300bbfeb0          BMOV.32.CLEAR R18, B6
0x0000004300bbfec0          MOV R4, R4
0x0000004300bbfed0          MOV R4, R4
0x0000004300bbfee0          MOV R17, R4
  return -1.0f + w1(a) / (w0(a) + w1(a)) + 0.5f;
0x0000004300bbfef0          MOV R4, R17
0x0000004300bbff00          MOV R20, 0x0
0x0000004300bbff10          MOV R21, 0x0
0x0000004300bbff20          CALL.ABS.NOINC 0x0
0x0000004300bbff30          MOV R0, R4
0x0000004300bbff40          MOV R4, R17
0x0000004300bbff50          MOV R16, R0
0x0000004300bbff60          MOV R20, 0x0
0x0000004300bbff70          MOV R21, 0x0
0x0000004300bbff80          CALL.ABS.NOINC 0x0
0x0000004300bbff90          MOV R0, R4
0x0000004300bbffa0          MOV R4, R17
0x0000004300bbffb0          MOV R2, R0
0x0000004300bbffc0          MOV R20, 0x0
0x0000004300bbffd0          MOV R21, 0x0
0x0000004300bbffe0          CALL.ABS.NOINC 0x0
0x0000004300bbfff0          MOV R4, R4
0x0000004300bc0000          FADD R4, R2, R4
0x0000004300bc0010          MOV R0, R16
0x0000004300bc0020          MOV R4, R4
0x0000004300bc0030          MOV R0, R0
0x0000004300bc0040          MOV R4, R4
0x0000004300bc0050          MOV R3, R4
0x0000004300bc0060          MUFU.RCP R5, R3
0x0000004300bc0070          FADD R3, -RZ, -R3
0x0000004300bc0080          MOV R3, R3
0x0000004300bc0090          MOV R6, 0x3f800000
0x0000004300bc00a0          FFMA R6, R3, R5, R6
0x0000004300bc00b0          FCHK P0, R0, R4
0x0000004300bc00c0          FFMA R6, R5, R6, R5
0x0000004300bc00d0          MOV R5, RZ
0x0000004300bc00e0          MOV R0, R0
```

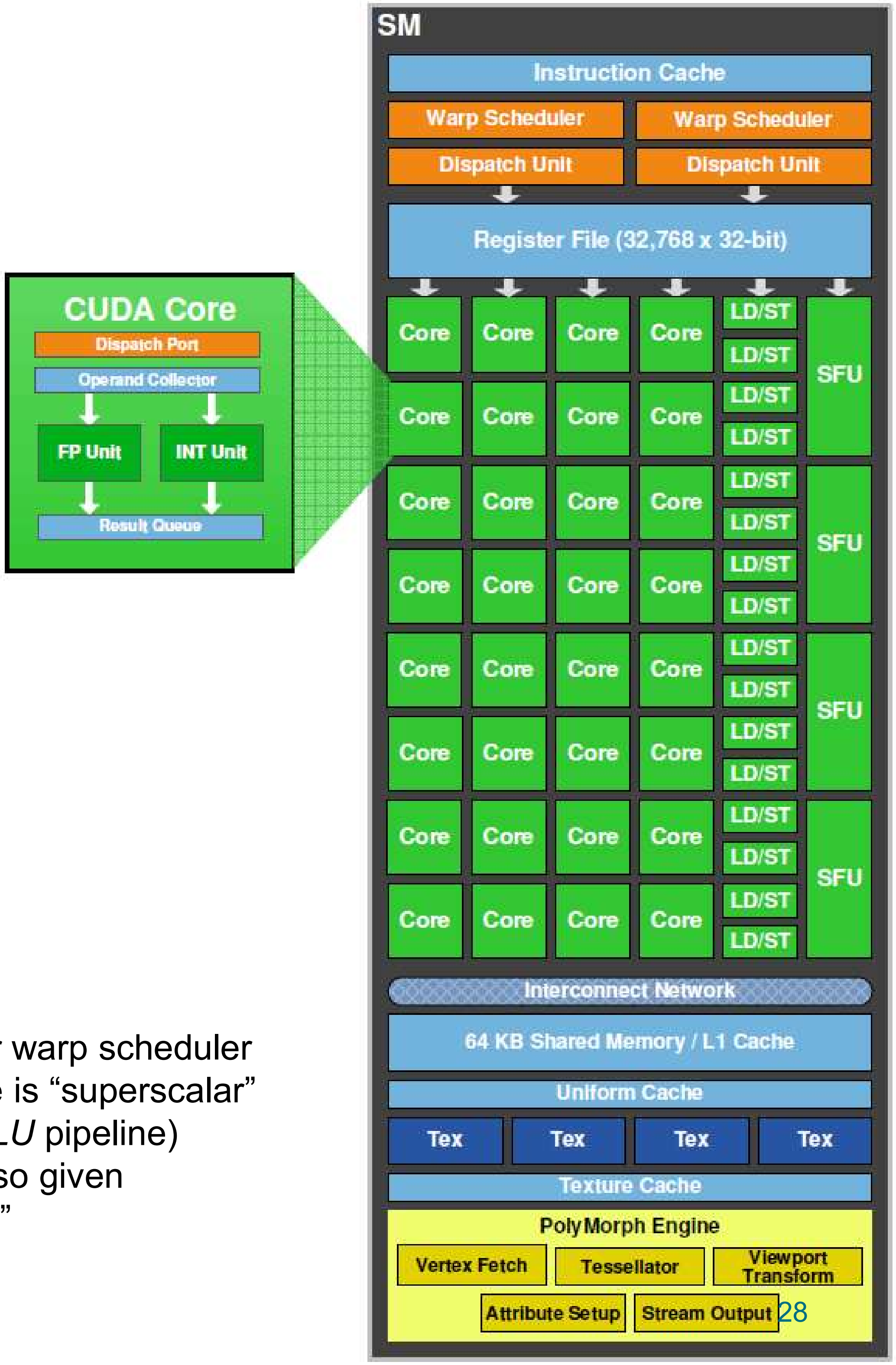
```
// h0 and h1 are the two offset functions
__device__ float h0(float a) {
  // note +0.5 offset to compensate for CUDA linear filtering convention
  return -1.0f + w1(a) / (w0(a) + w1(a)) + 0.5f;
}

__device__ float h1(float a) { return 1.0f + w3(a) / (w2(a) + w3(a)) + 0.5f; }
```

Example: “Scalar” GF100

Main concept here:

There is one instruction dispatcher
(dispatch unit / fetch/decode unit)
per warp scheduler
(warp selector)

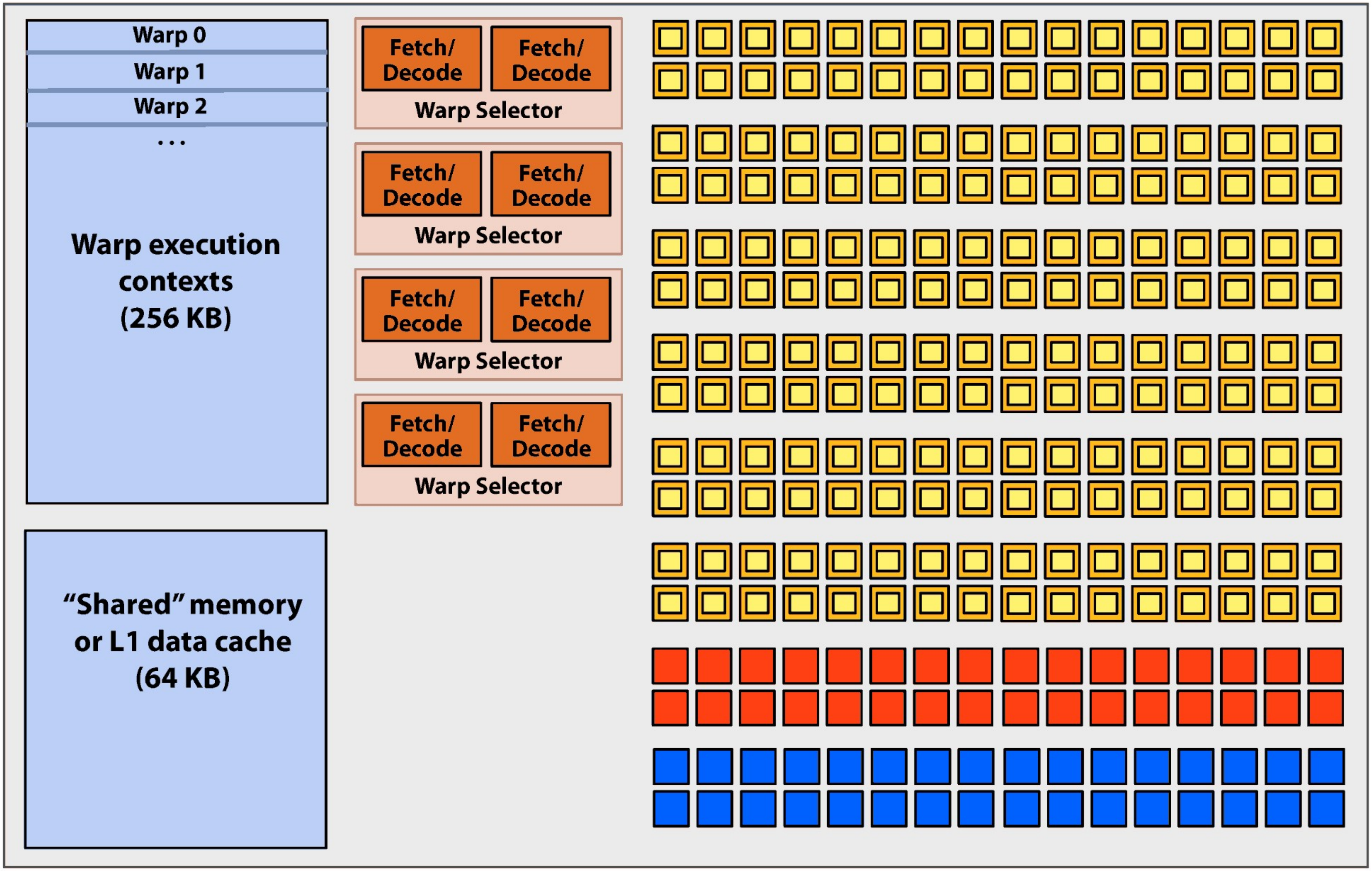


Details later...
Ignore less important subtleties...
GF100 has two warp schedulers, not one,
and each 32-thread instruction is executed
over two clock cycles, not one, etc.

Caveat on NVIDIA diagrams: if two dispatchers per warp scheduler are shown, it still doesn‘t mean that the ALU pipeline is “superscalar” (often, the second dispatcher dispatches to a *non-ALU* pipeline)
... need to look at CUDA programming guide info, also given
in our tables in row “*# ALU dispatch / warp sched.*”

Example: “Superscalar” ALUs in SM Architecture

NVIDIA Kepler GK104 architecture SMX unit (one “core”)



Warp 0

Warp 1

Warp 2

. . .

Warp execution contexts (256 KB)

“Shared” memory or L1 data cache (64 KB)

Fetch/ Decode | Fetch/ Decode | Warp Selector

Fetch/ Decode | Fetch/ Decode | Warp Selector

Fetch/ Decode | Fetch/ Decode | Warp Selector

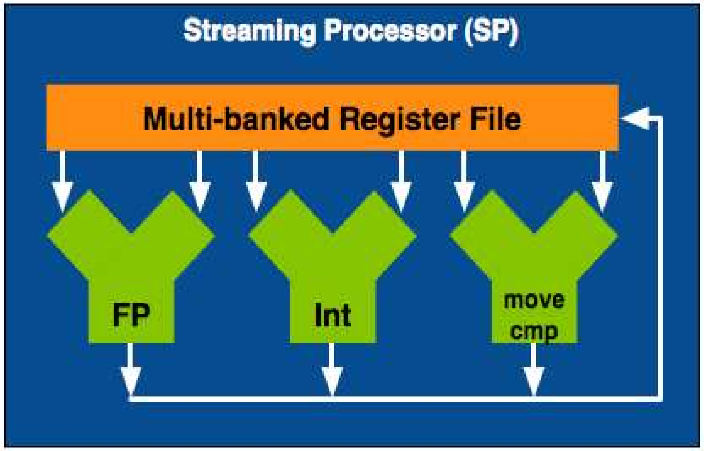
Fetch/ Decode | Fetch/ Decode | Warp Selector

🟨 = SIMD function unit,
control shared across 32 units
(1 MUL-ADD per clock)

🟥 = “special” SIMD function unit,
control shared across 32 units
(operations like sin/cos)

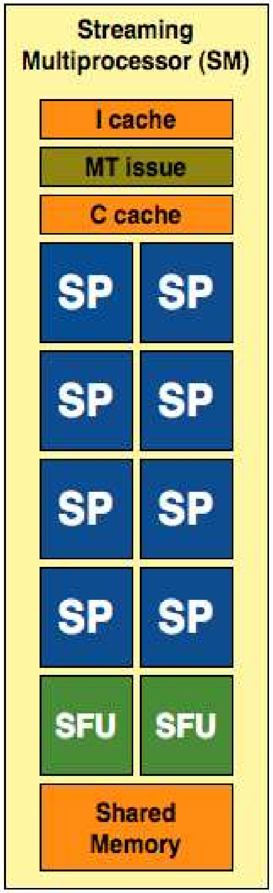
🟦 = SIMD load/store unit
(handles warp loads/stores, gathers/scatters)

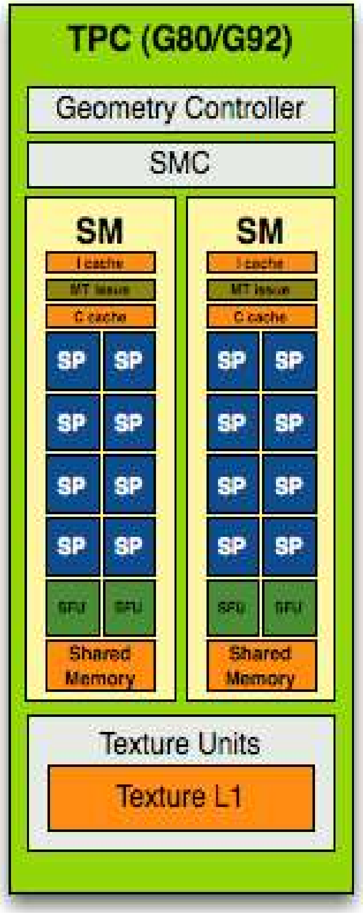
NVIDIA Tesla Architecture (not the Tesla product line!), G80: 2007, GT200: 2008/2009

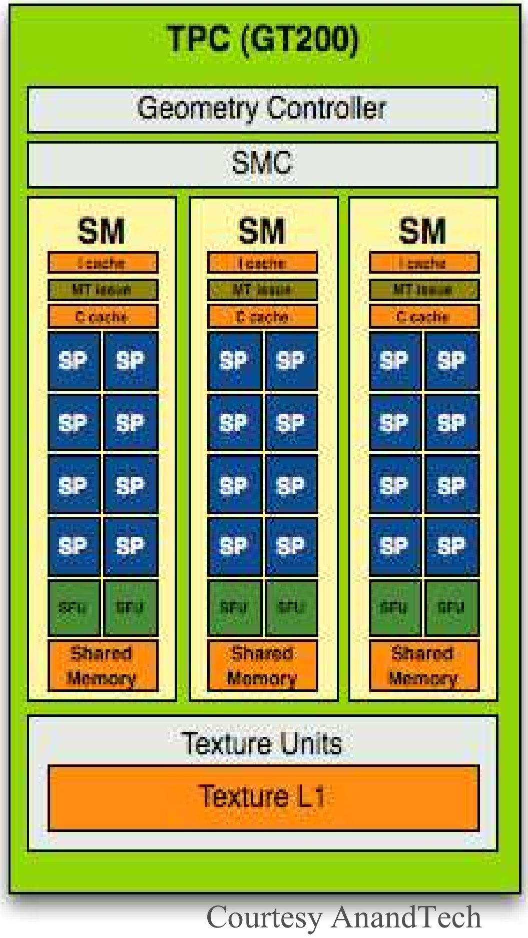


G80: first CUDA GPU!

Multiprocessor: SM (CC 1.x)





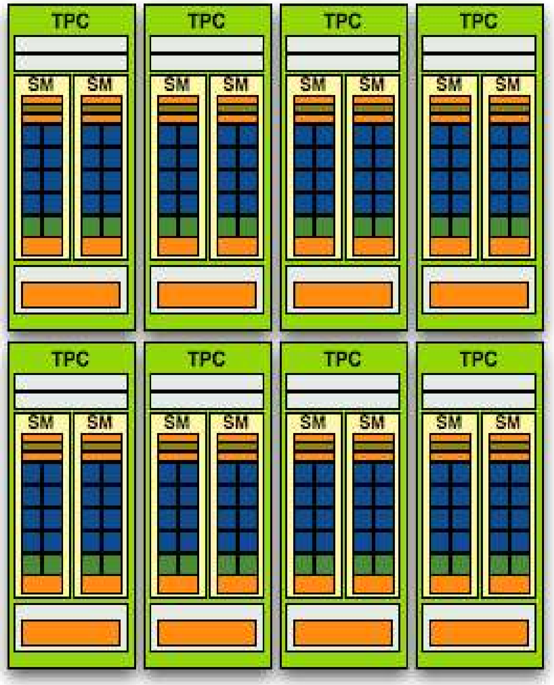


Courtesy AnandTech

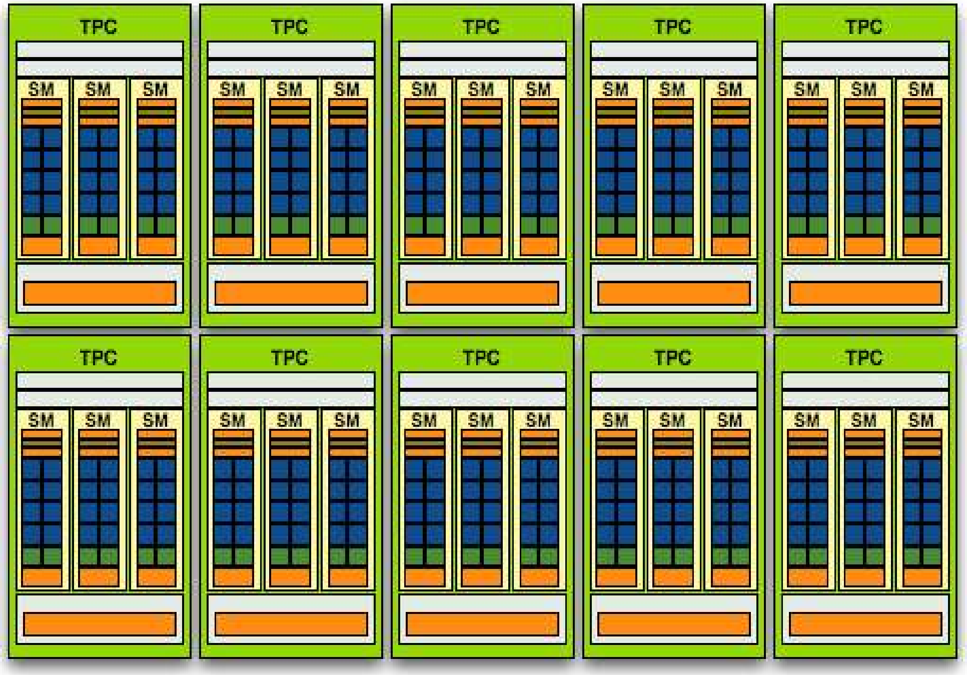
- Streaming Processor (SP) [or: CUDA core; or: FP32 / FP64 / INT32 core, ...]
- Streaming Multiprocessor (SM)
- Texture/Processing Cluster (TPC)

NVIDIA Tesla Architecture (not the Tesla product line!), G80: 2007, GT200: 2008/2009

- G80/G92: 8 TPCs * (2 * 8 SPs) = 128 SPs [= CUDA cores]
- GT200: 10 TPCs * (3 * 8 SPs) = 240 SPs [= CUDA cores]
- **Arithmetic intensity** has increased (num. of ALUs vs. texture units)



G80 / G92

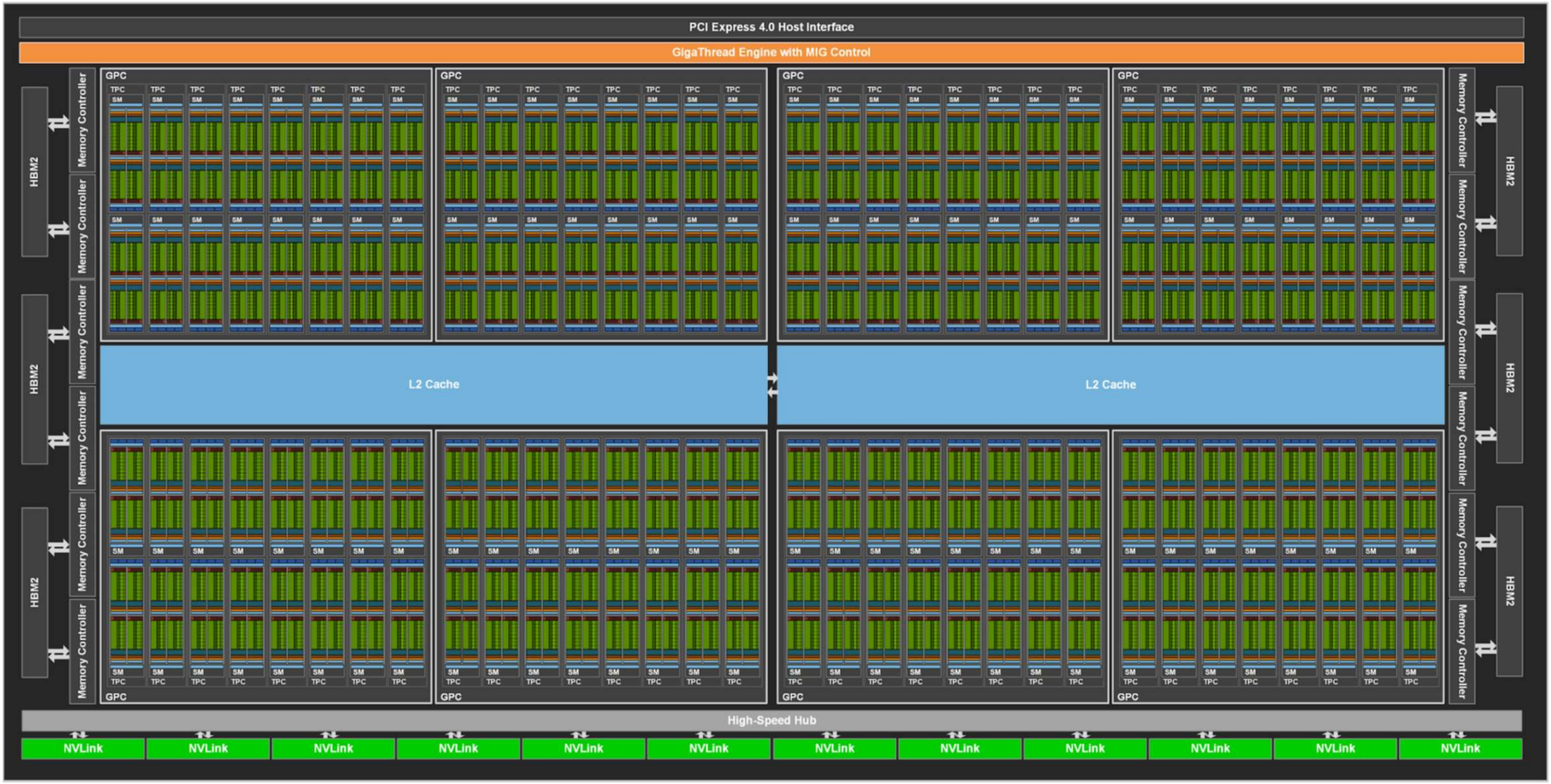


GT200

Courtesy AnandTech

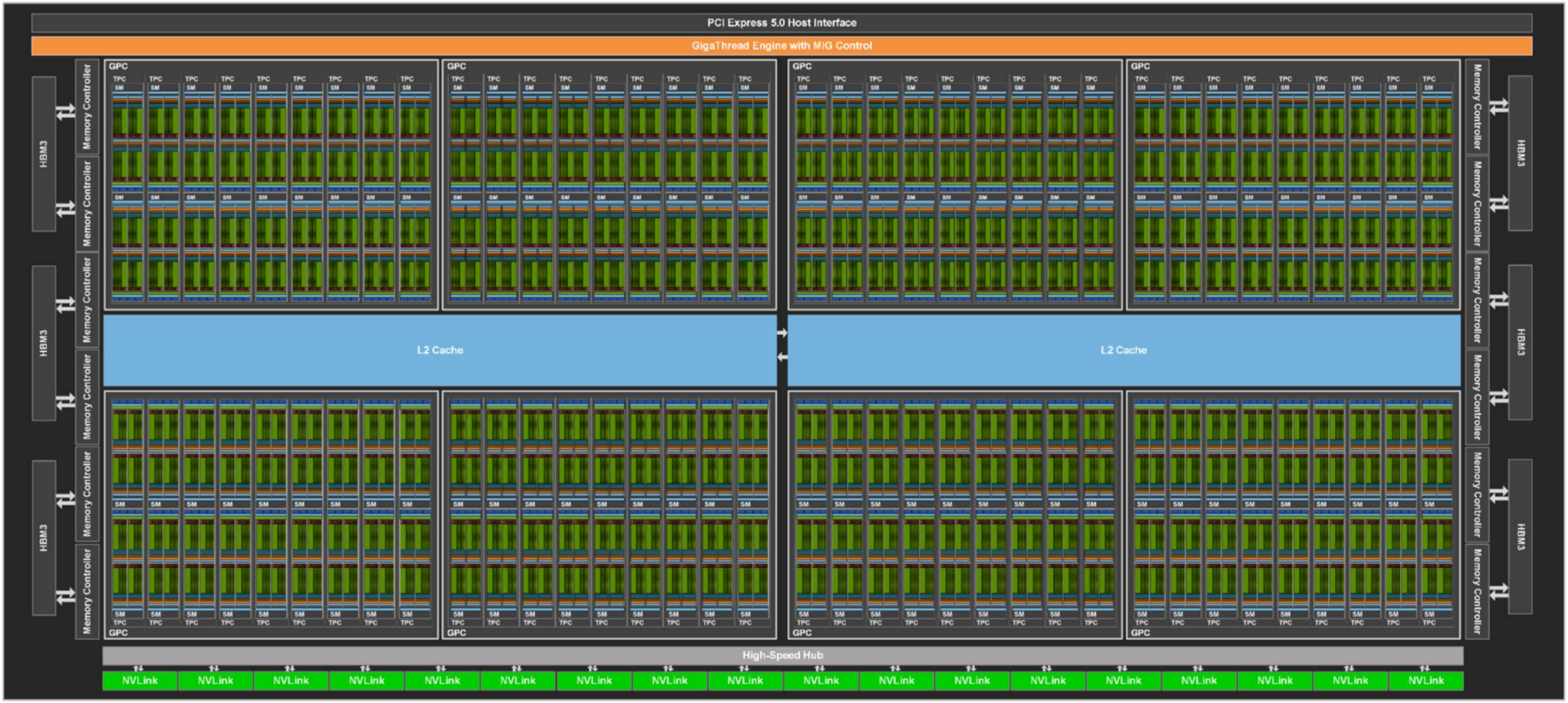
NVIDIA Ampere GA100 Architecture (2020)

GA 100 (A100 Tensor Core GPU) Full GPU: 128 SMs (in 8 GPCs/64 TPCs)


PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

NVIDIA Hopper GH100 Architecture (2022)

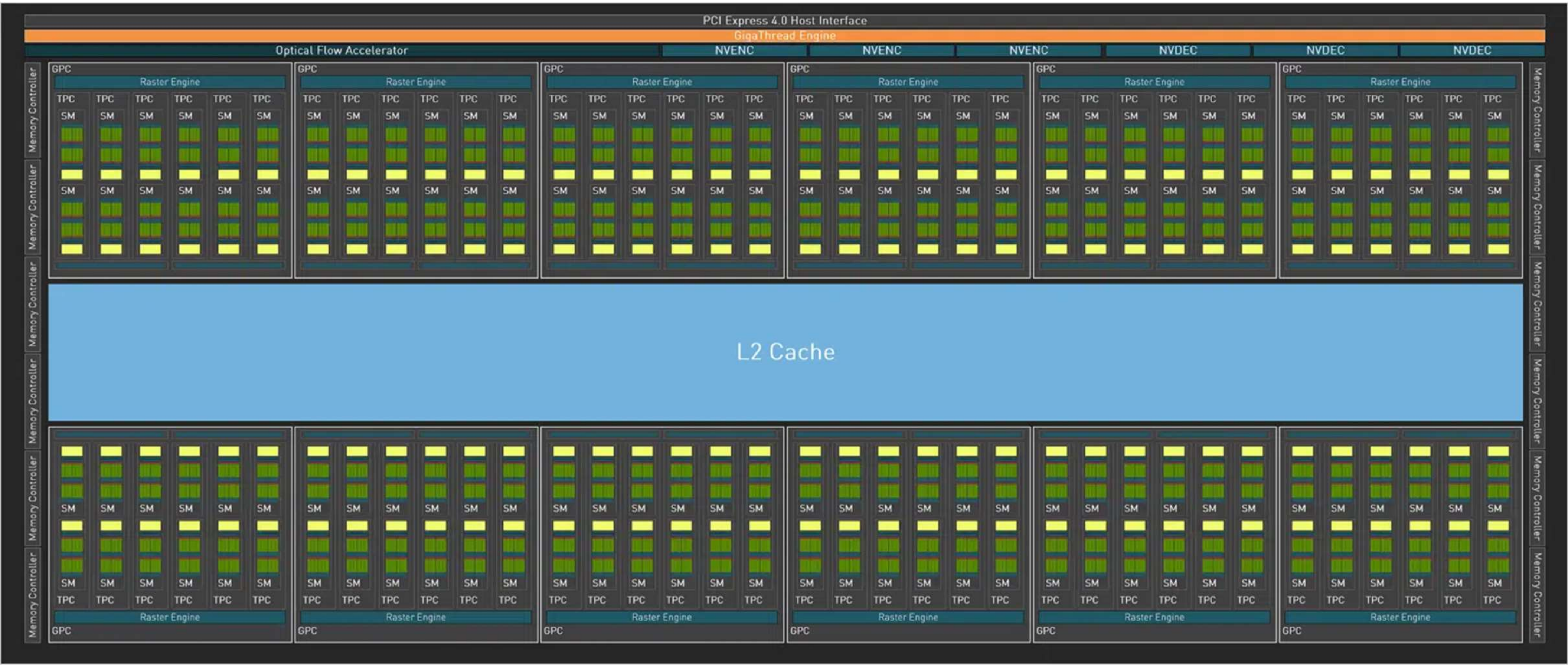
GH 100 (H100 Tensor Core GPU)   Full GPU: 144 SMs (in 8 GPCs/72 TPCs)



NVIDIA Ada Lovelace AD10x Architecture (2022)

Full AD 10x

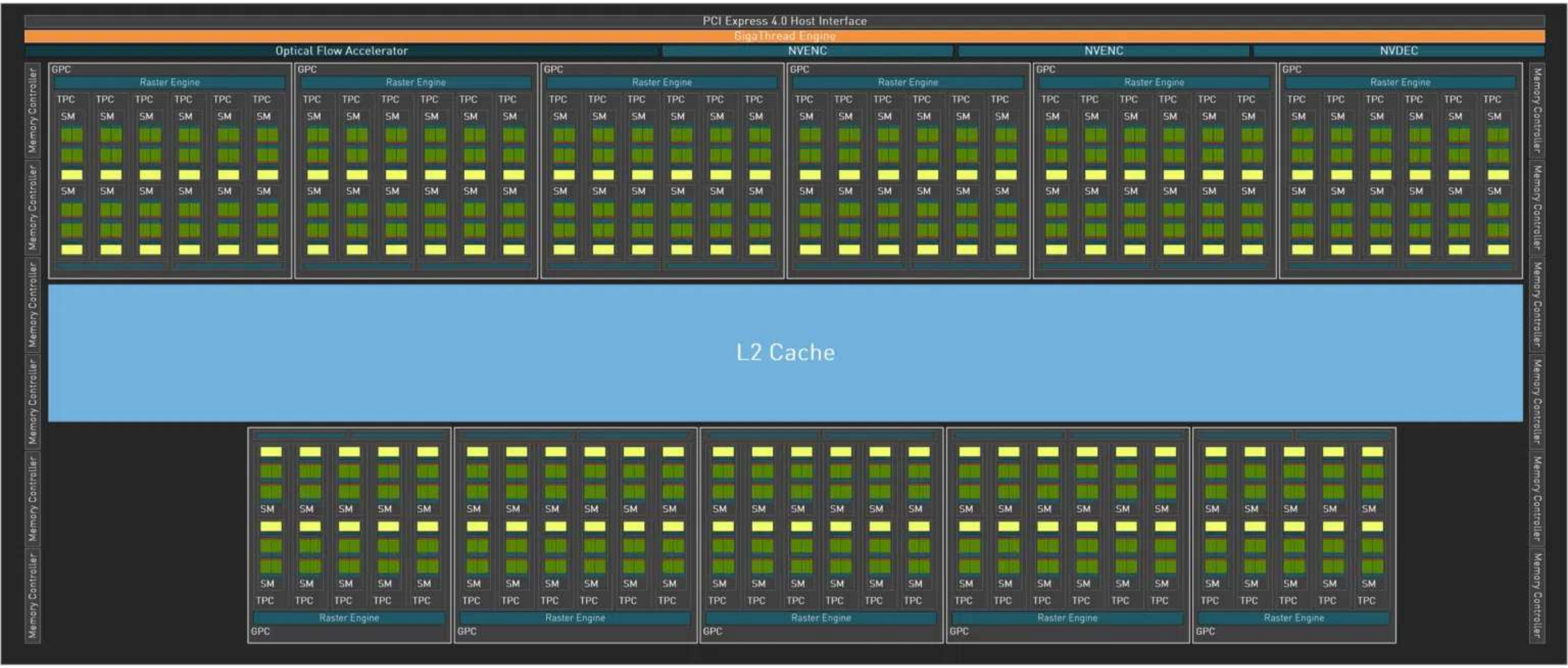
Full GPU: 144 SMs (in 12 GPCs/72 TPCs)



NVIDIA Ada Lovelace AD102 Architecture (2022)

AD 102 (RTX 4090, …)

Full RTX 4090: 128 SMs (in 11 GPCs/64 TPCs)



Thank you.