# PROFILING & OPTIMIZING GPU KERNELS

Vishal Mehta

# BEFORE YOU START

## The five steps to enlightenment

1. Know your application
   - What does it compute? How is it parallelized? What final performance is expected?
2. Know your hardware
   - What are the target machines, how many nodes? Machine-specific optimizations okay?
3. Know your tools
   - Strengths and weaknesses of each tool? Learn how to use them (and learn one well!)
4. Know your process
   - Performance optimization is a constant learning process
5. Make it so!

# THE APOD CYCLE

4. Deploy and Test

1. Assess

- Identify Performance Limiter
- Analyze Profile
- Find Indicators

3. Optimize

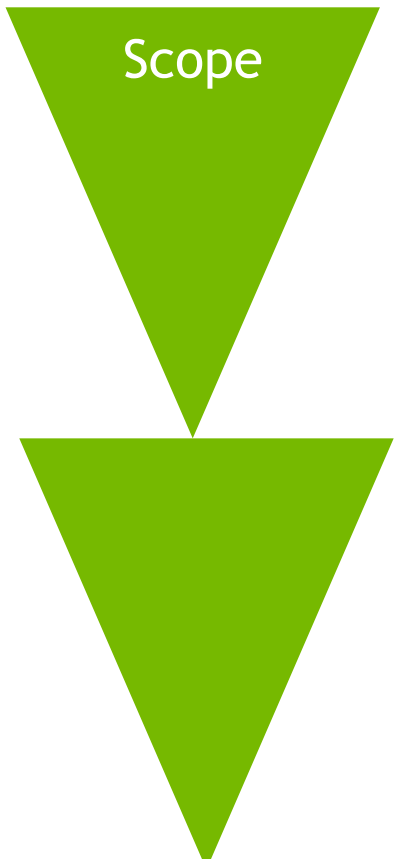
3b. Build Knowledge

2. Parallelize

# GUIDING OPTIMIZATION EFFORT

## "Drilling Down into the Metrics"

- Challenge: How to know where to start?
- Top-down Approach:
  - Find Hotspot Kernel
  - Identify Performance Limiter of the Hotspot
  - Find performance bottleneck indicators related to the limiter
  - Identify associated regions in the source code
  - Come up with strategy to fix and change the code
  - Start again

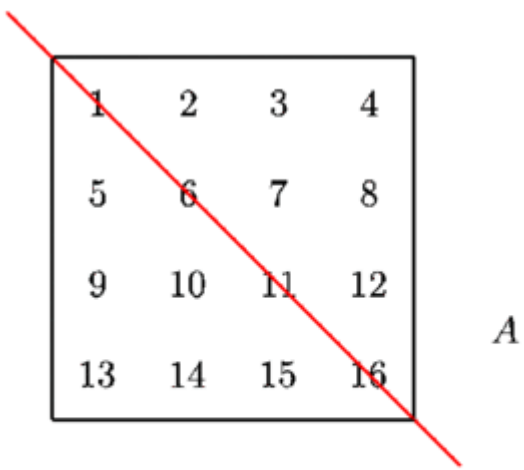


# KNOW YOUR APPLICATION: MATRIX TRANSPOSE

# MATRIX TRANSPOSE

## Parallel transpose kernel

- Motivation: Used in lot of Linear Algebra, Relatively easy for hands-on, but still a lot that could be done to optimize




```
// Simplest transpose; doesn't use shared memory.
// Global memory reads are coalesced but writes are not.
__global__ void transposeNaive(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[x*width + (y+j)] = idata[(y+j)*width + x];
}
```

# KNOW YOUR HARDWARE: PASCAL ARCHITECTURE

# GPU COMPARISON

| | V100 | P100 (SXM2) | M40 | K40 |
|---|---|---|---|---|
| Double/Single/Half TFlop/s | **7.8/15.7/125 (TensorCore)** | **5.3/10.6/21.2** | 0.2/7.0/NA | 1.4/4.3/NA |
| Memory Bandwidth (GB/s) | **900** | **732** | 288 | 288 |
| Memory Size | **16/32 GB** | **16GB** | 12GB, 24GB | 12GB |
| L2 Cache Size | **6144 KB** | **4096 KB** | 3072 KB | 1536 KB |
| Base/Boost Clock (Mhz) | **1312 / 1530** | **1328/1480** | 948/1114 | 745/875 |
| TDP (Watts) | **300** | **300** | 250 | 235 |

# GP100 SM

| | GP100 |
|---|---|
| CUDA Cores | 64 |
| Register File | 256 KB |
| Shared Memory | 64 KB |
| Active Threads | 2048 |
| Active Blocks | 32 |

# KNOW YOUR TOOLS: PROFILERS

# PROFILING TOOLS

## Many Options!

### From NVIDIA

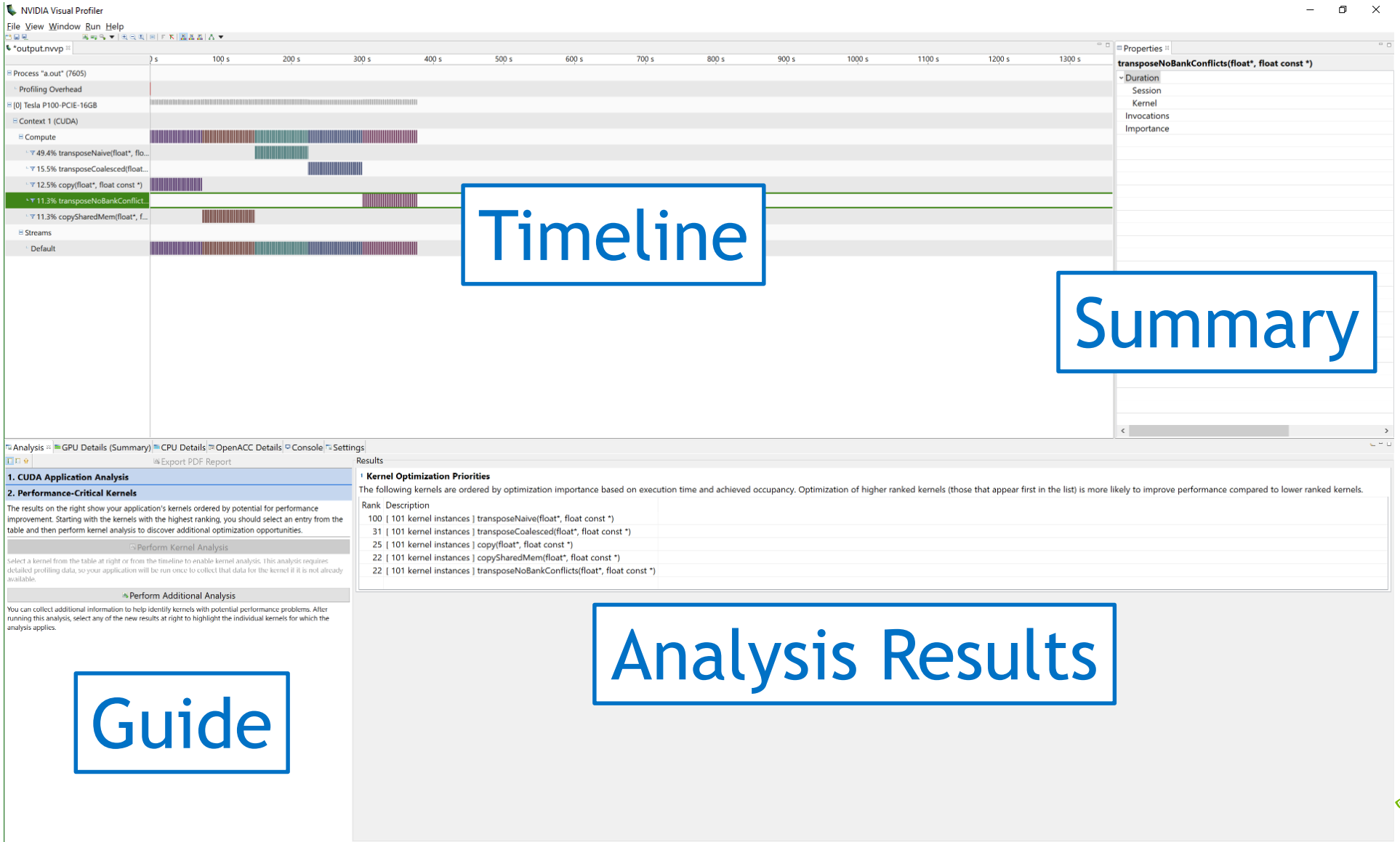
- nvprof
- NVIDIA Visual Profiler
  - Standalone (nvvp)
  - Integrated into Nsight Eclipse Edition (nsight)
- Nsight Visual Studio Edition

### Third Party

- TAU Performance System
- VampirTrace
- PAPI CUDA component
- HPC Toolkit
- (Tools using CUPTI)

In this session we will be showing nvvp screenshots

# THE NVVP PROFILER WINDOW



# MAKE IT HAPPEN: ITERATION 1

## GLOBAL MEMORY COALESCING

# IDENTIFY HOTSPOT

Hotspot

Results

**Kernel Optimization Priorities**

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

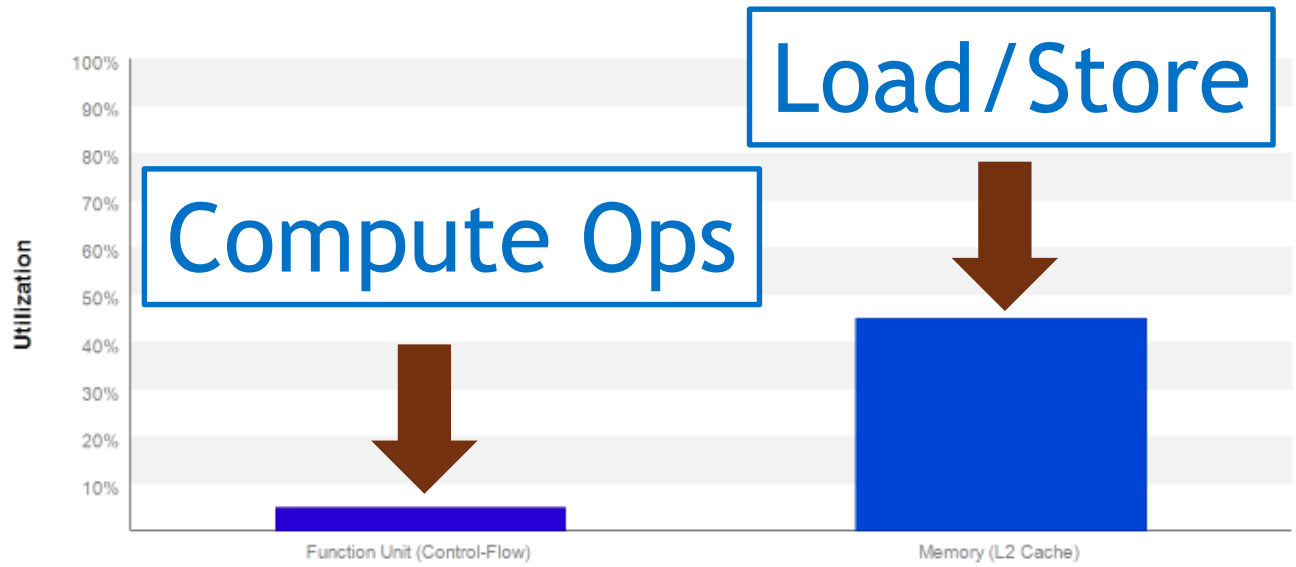
| Rank | Description |
|---|---|
| 100 | [ 101 kernel instances ] transposeNaive(float*, float const *) |
| 31 | [ 101 kernel instances ] transposeCoalesced(float*, float const *) |
| 25 | [ 101 kernel instances ] copy(float*, float const *) |
| 22 | [ 101 kernel instances ] copySharedMem(float*, float const *) |
| 22 | [ 101 kernel instances ] transposeNoBankConflicts(float*, float const *) |

# IDENTIFY PERFORMANCE LIMITER

Results

**Kernel Performance Is Bound By Instruction And Memory Latency**

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Tesla P100-PCIE-16GB". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

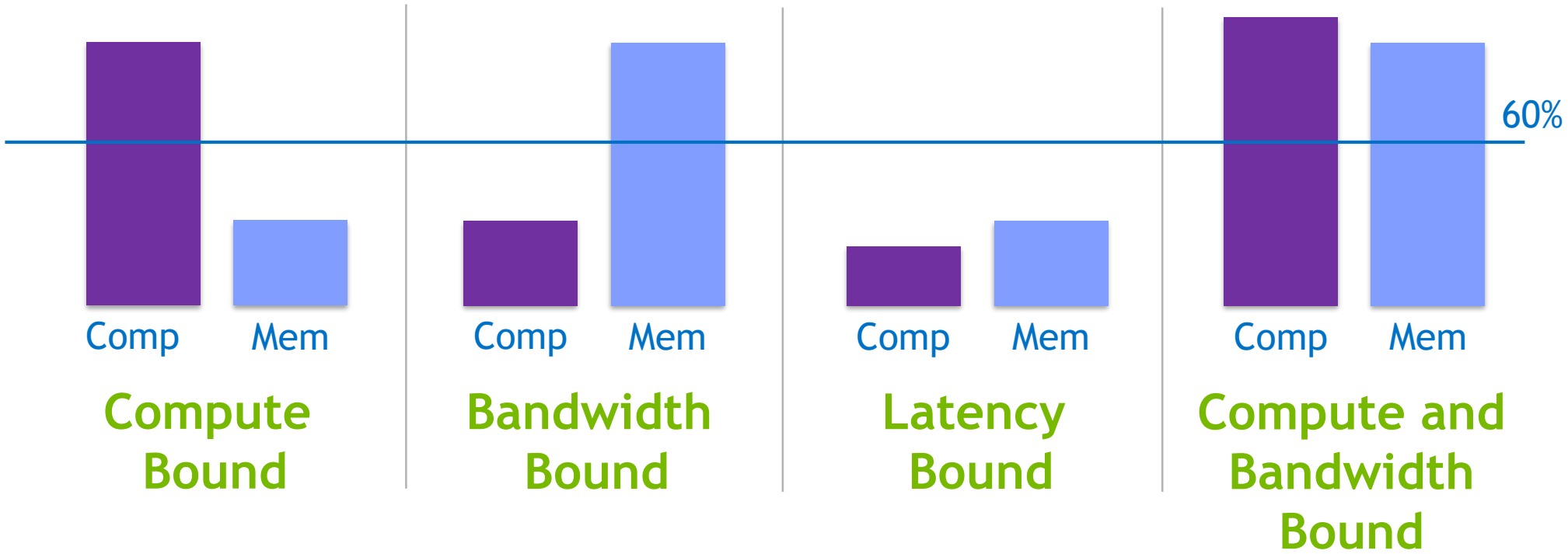



Memory Utilization Issues?

# PERFORMANCE LIMITER CATEGORIES

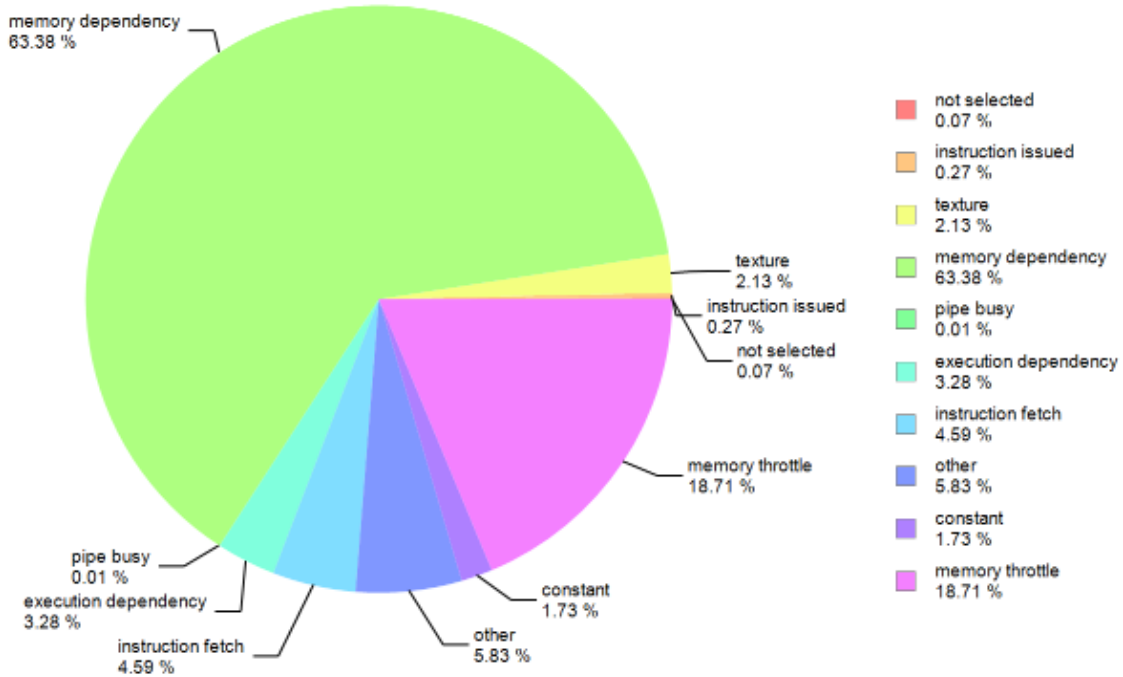
Memory Utilization vs Compute Utilization

Four possible combinations:



# DRILLING DOWN: LATENCY ANALYSIS

| | |
|---|---|
| Grid Size | [ 65536,1,1 ] |
| Block Size | [ 8,4,1 ] |
| ◢ Occupancy | |
| Achieved | ⚠ 49.7% |
| Theoretical | 50% |
| Limiter | Block Size |




**⚠ GPU Utilization May Be Limited By Block Size**

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

# OCCUPANCY

## GPU Utilization

Each SM has limited resources:

- max. 64K Registers (32 bit) distributed between threads
- max. 48KB of shared memory per block (96KB per SMM)
- max. 32 Active Blocks per SMM
- Full occupancy: 2048 threads per SM (64 warps)

When a resource is used up, occupancy is reduced

*(*) Values vary with Compute Capability*

# LOOKING FOR MORE INDICATORS

Line Global Access File - /C:/Users/vishalm/Desktop/transpose_result.cu

```
87   tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] = idata[(y+j)*width + x];
88
89   __syncthreads();
90
91   for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
92     odata[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x];
93 }
94
95 // naive transpose
96 // Simplest transpose; doesn't use shared memory.
97 // Global memory reads are coalesced but writes are not.
98 __global__ void transposeNaive(float *odata, const float *idata)
99 {
100  int x = blockIdx.x * TILE_DIM + threadIdx.x;
101  int y = blockIdx.y * TILE_DIM + threadIdx.y;
102  int width = gridDim.x * TILE_DIM;
103
104  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
105    odata[x*width + (y+j)] = idata[(y+j)*width + x];
106 }
107
108 // coalesced transpose
109 // Uses shared memory to achieve coalesing in both reads and writes
110 // Tile width == #banks causes shared memory bank conflicts.
111 __global__ void transposeCoalesced(float *odata, const float *idata)
112 {
```

Global Access Disassembly

```
SHL R5, R7, 0x8 ;
SHL R4, R0.reuse, 0x2 ;
SHR R0, R0, 0x1e ;
SHL R8, R5, 0x2 ;
IADD R6.CC, R4, c[0x0][0x140] ;
SHR R9, R5, 0x1e ;
IADD.X R0, R0, c[0x0][0x144] ;
IADD R4.CC, R8, R2 ;
MOV R2, R6 ;
IADD.X R5, R9, R3 ;
MOV R3, R0 ;
STG.E [R2], R11 ;
LDG.E R0, [R4] ;
IADD R6.CC, R8, R4 ;
IADD.X R12, R9, R5 ;
MOV R7, R12 ;
STG.E [R2+0x20], R0 ;
LDG.E R7, [R6] ;
IADD R8.CC, R8, R6 ;
IADD.X R9, R9, R12 ;
STG.E [R2+0x40], R7 ;
LDG.E R8, [R8] ;
STG.E [R2+0x60], R8 ;
EXIT ;
.L_1:
```

Select or highlight a single interval to see properties

Source Code Association

Analysis | GPU Details (Summary) | CPU Details | OpenACC Details | Console | Settings

transposeNaive(float*, float const *)

- Kernel Performance Limiter
- Kernel Latency
- Kernel Compute
- Kernel Memory

Results

**Global Memory Alignment and Access Pattern**

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern. The analysis is per assembly instruction.

*Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.* More...

| Line / File | transpose_result.cu - \users\vmehta\summerschool |
|---|---|
| 105 | Global Store L2 Transactions/Access = 32, Ideal Transactions/Access = 4 [ 262144 L2 transactions for 8192 total executions ] |
| 105 | Global Store L2 Transactions/Access = 32, Ideal Transactions/Access = 4 [ 262144 L2 transactions for 8192 total executions ] |
| 105 | Global Store L2 Transactions/Access = 32, Ideal Transactions/Access = 4 [ 262144 L2 transactions for 8192 total executions ] |
| 105 | Global Store L2 Transactions/Access = 32, Ideal Transactions/Access = 4 [ 262144 L2 transactions for 8192 total executions ] |

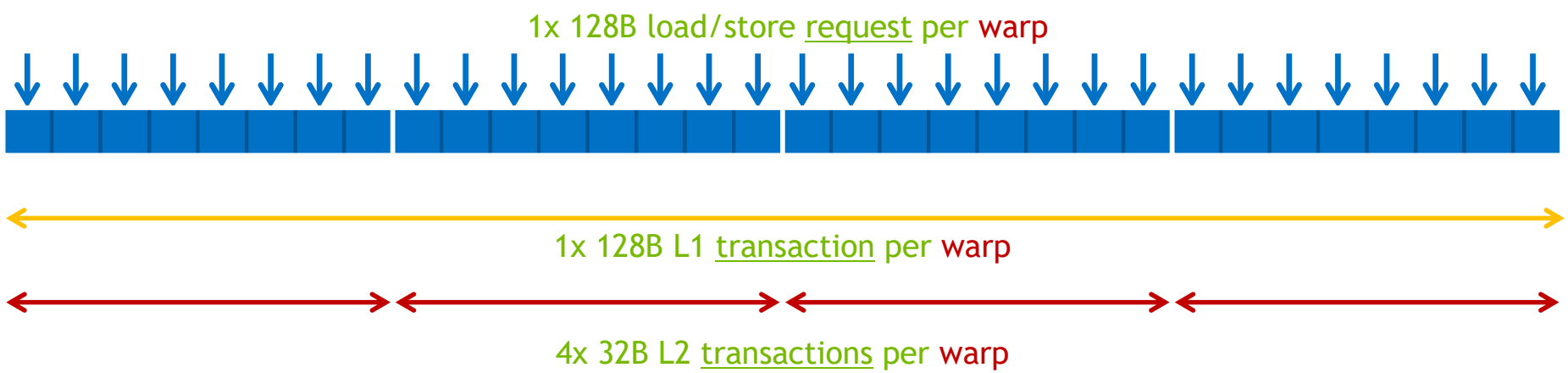
For line numbers use:
`nvcc -lineinfo`

**32 Global STORE Transactions**

# MEMORY TRANSACTIONS: BEST CASE

A warp issues 32x4B aligned and consecutive load/store request

Threads read different elements of the same 128B segment



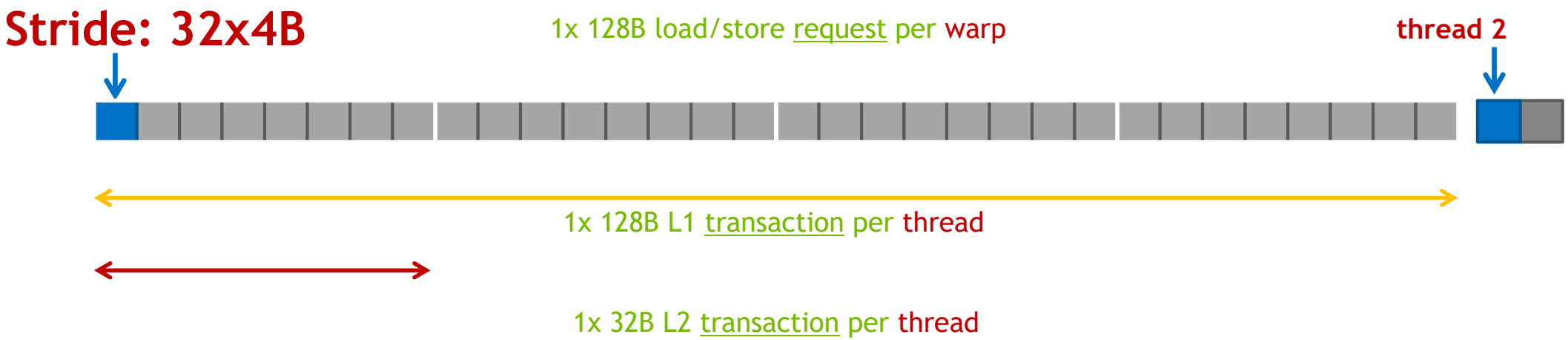

1x L1 transaction: 128B needed / 128B transferred

4x L2 transactions: 128B needed / 128B transferred

# MEMORY TRANSACTIONS: WORST CASE

Threads in a warp read/write 4B words, 128B between words

Each thread reads the first 4B of a 128B segment



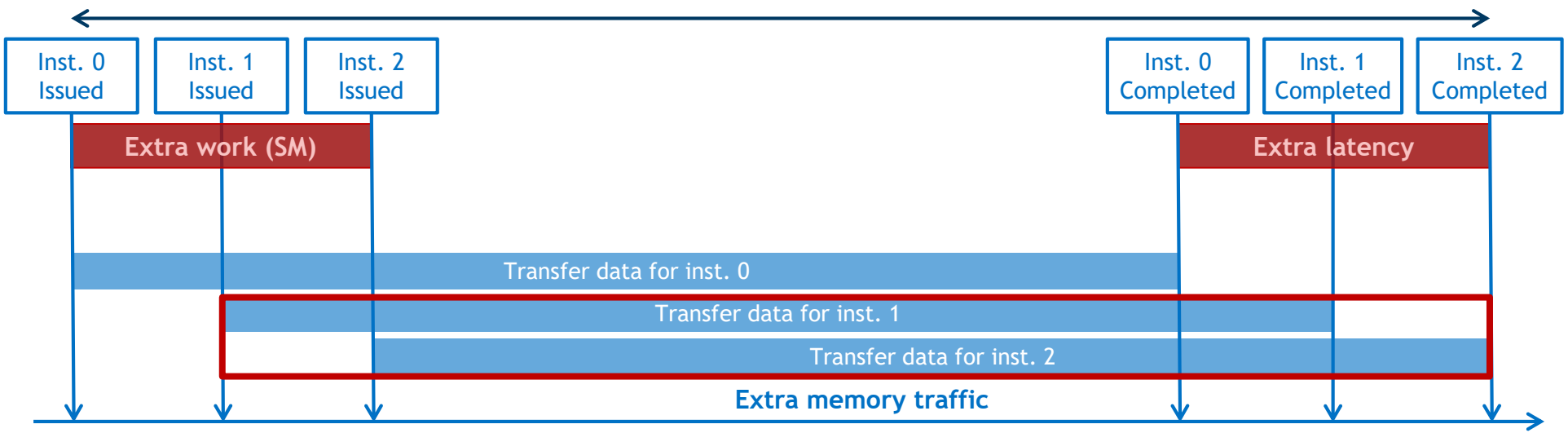

32x L1 transactions: 128B needed / 32x 128B transferred

32x L2 transactions: 128B needed / 32x 32B transferred

# TRANSACTIONS AND REPLAYS

With replays, requests take more time and use more resources

- More instructions issued
- More memory traffic
- Increased execution time



# PERF-OPT QUICK REFERENCE CARD

| Category: | Latency Bound - Occupancy |
|---|---|
| Problem: | Latency is exposed due to low occupancy |
| Goal: | **Hide** latency behind more parallel work |
| Indicators: | Occupancy low (< 60%)<br>Execution Dependency High |
| Strategy: | Increase occupancy by:<br>• Varying block size<br>• Varying shared memory usage<br>• Varying register count (use __launch_bounds) |

# PERF-OPT QUICK REFERENCE CARD

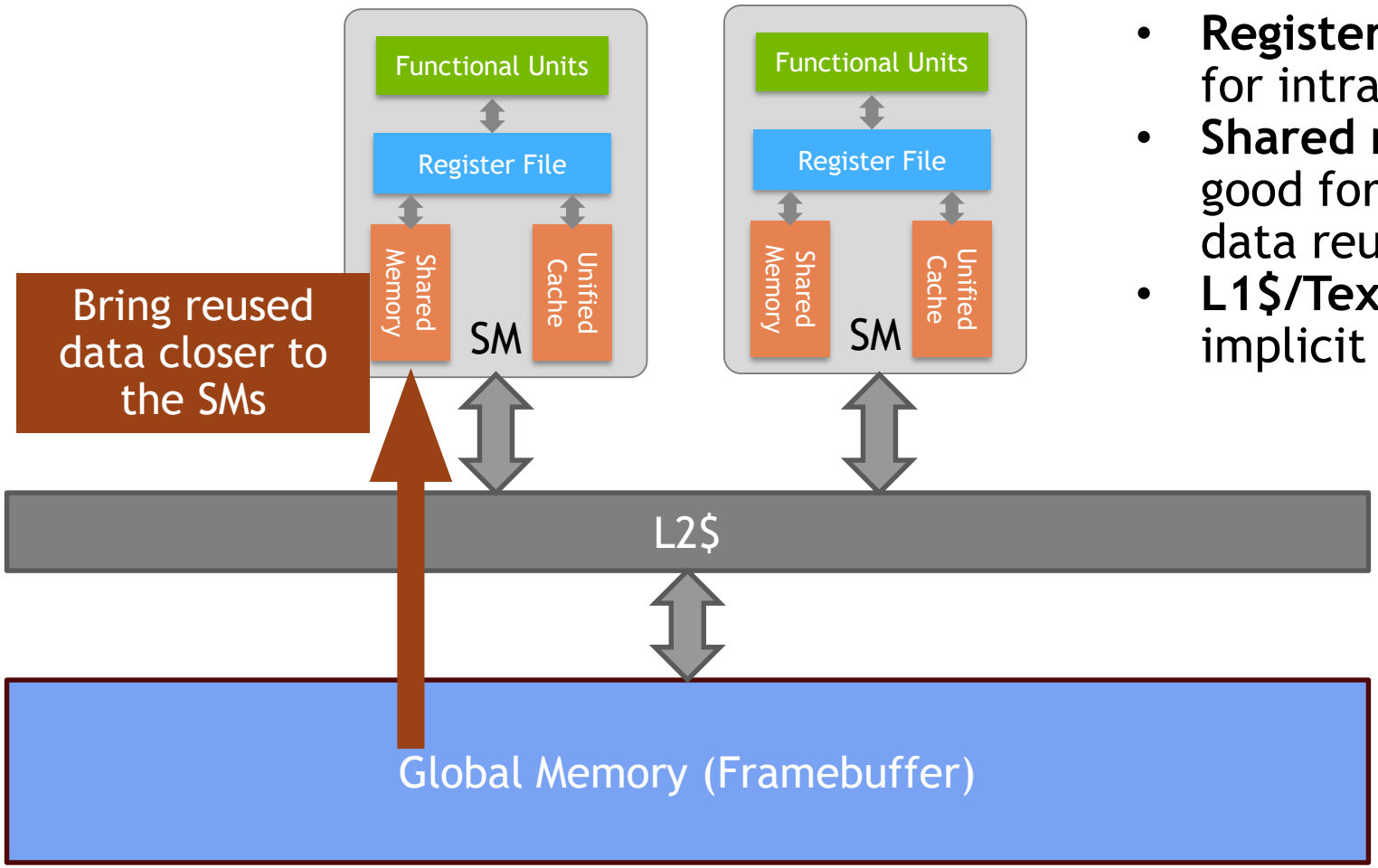
| Category: | Latency Bound - Coalescing |
|---|---|
| Problem: | Memory is accessed inefficiently => high latency |
| Goal: | Reduce #transactions/request to reduce latency |
| Indicators: | Low global load/store efficiency,<br>High #transactions/#request compared to ideal |
| Strategy: | Improve memory coalescing by:<br>• Cooperative loading inside a block<br>• Change block layout<br>• Aligning data<br>• Changing data layout to improve locality |

# PERF-OPT QUICK REFERENCE CARD

| Category: | Bandwidth Bound - Coalescing |
|---|---|
| Problem: | Too much unused data clogging memory system |
| Goal: | Reduce traffic, move more useful data per request |
| Indicators: | Low global load/store efficiency,<br>High #transactions/#request compared to ideal |
| Strategy: | Improve memory coalescing by:<br>• Cooperative loading inside a block<br>• Change block layout<br>• Aligning data<br>• Changing data layout to improve locality |

# GPU MEMORY HIERARCHY

## P100 (SMX2)

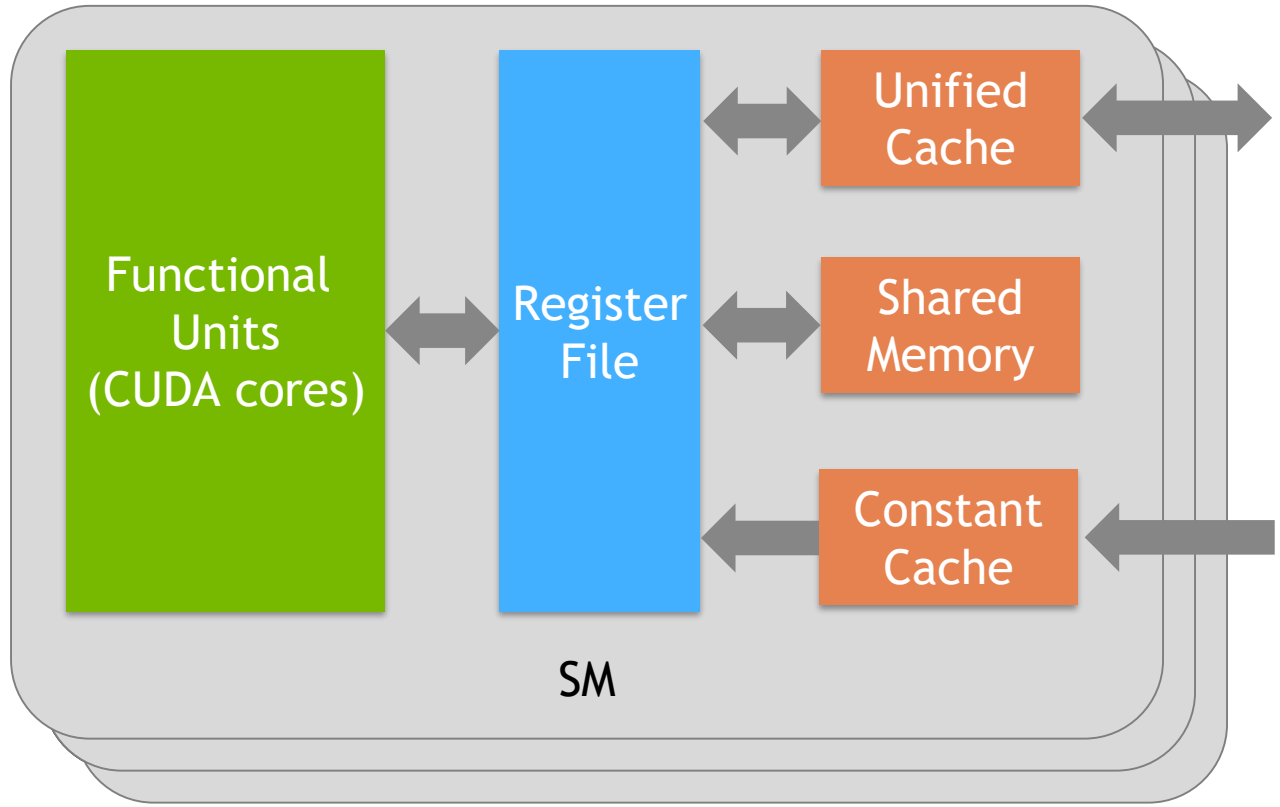



- **Registers (256 KB/SM):** good for intra-thread data reuse
- **Shared memory (64 KB/SM):** good for explicit intra-block data reuse
- **L1$/Tex$, L2$ (4096 KB):** implicit data reuse

# GPU SM ARCHITECTURE

## Pascal SM

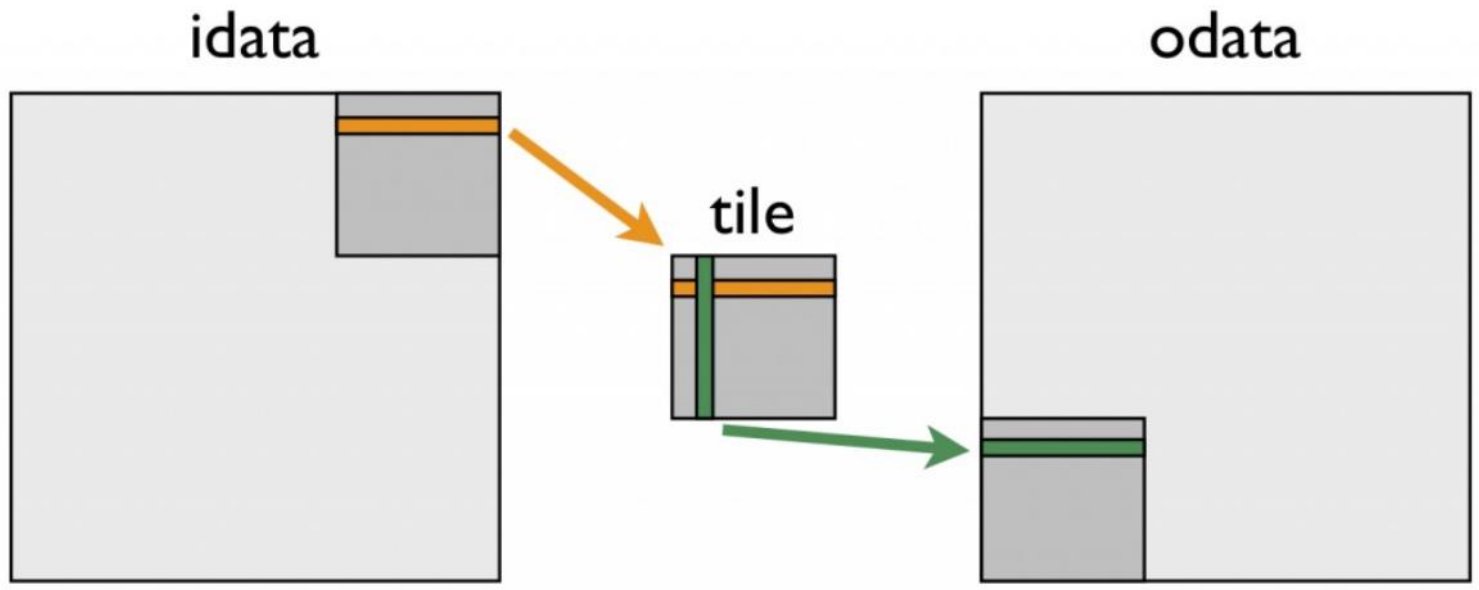
| | GP100 |
|---|---|
| CUDA Cores | 64 |
| Register File | 256 KB |
| Shared Memory | 64 KB |



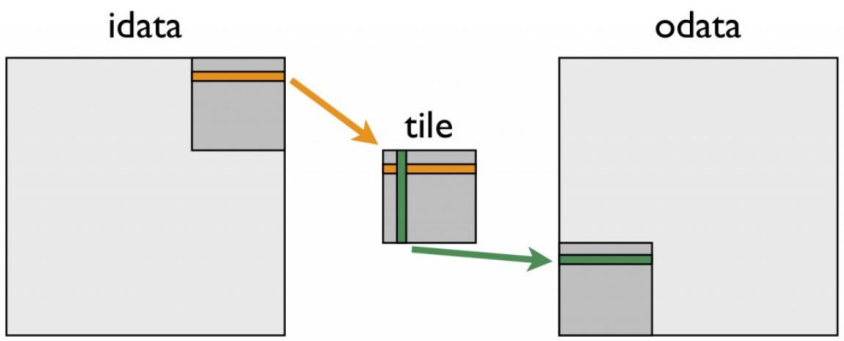

56 SMs on Tesla P100

# USING SHARED MEMORY IN MATRIX TRANSPOSE

# TILED MATRIX TRANSPOSE



# TILED MATRIX TRANSPOSE



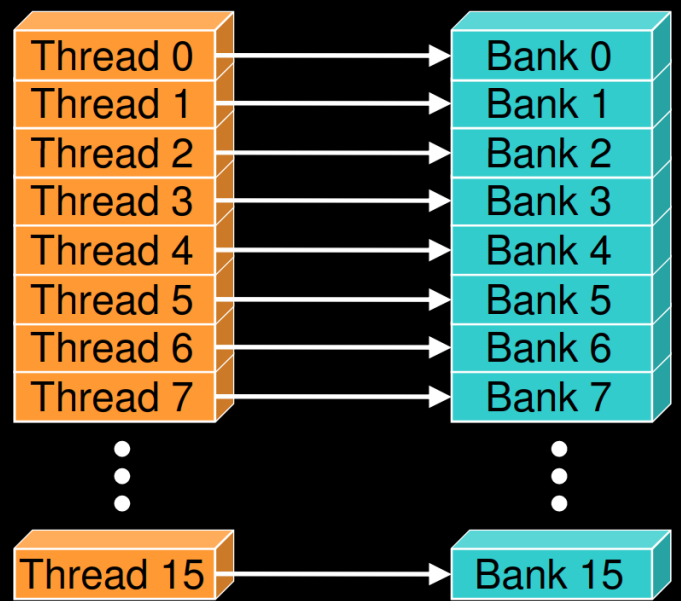

```
Device : Tesla P100-PCIE-16GB
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
dimGrid: 32 32 1. dimBlock: 32 8 1
                  Routine        Bandwidth (GB/s)
                     copy              442.94
       shared memory copy              472.60
          naive transpose              111.13
      coalesced transpose              363.88
```

```
// coalesced transpose
// Uses shared memory to achieve coalesing in both reads and writes
// Tile width == #banks causes shared memory bank conflicts.
__global__ void transposeCoalesced(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

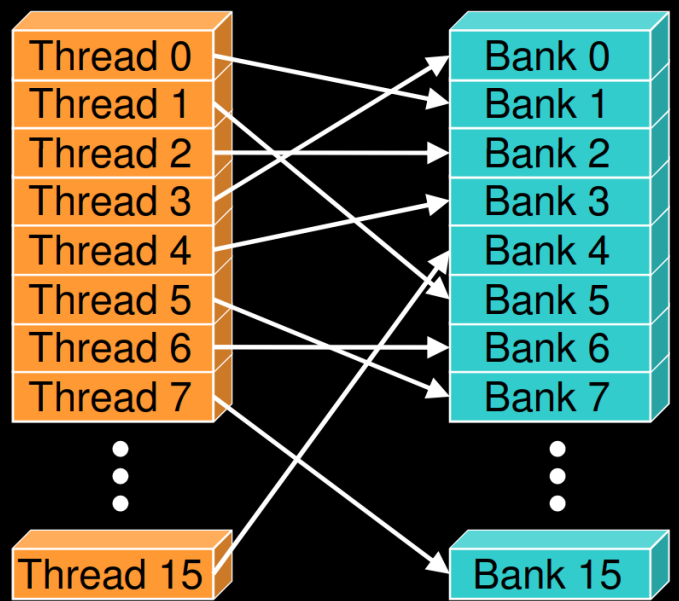
# ITERATION 2:
# SHARED MEMORY OPTIMIZATION

# SHARED MEMORY EXAMPLES

- No Bank Conflicts
  - Linear addressing stride == 1

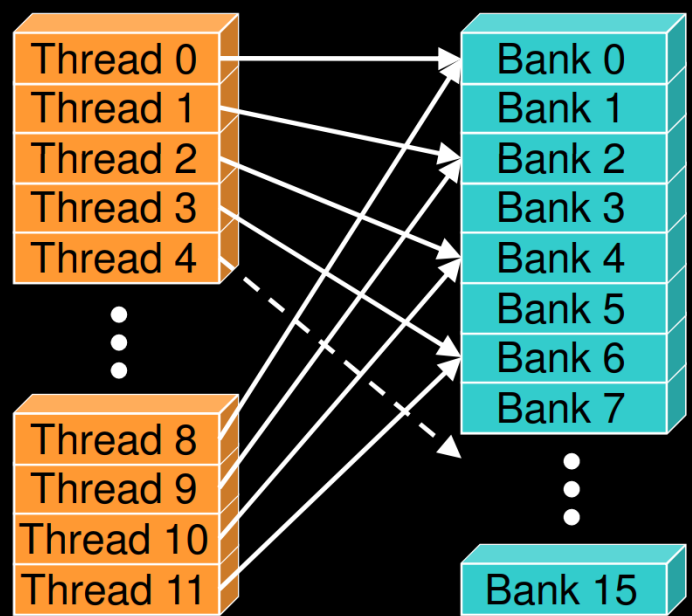



- No Bank Conflicts
  - Random 1:1 Permutation

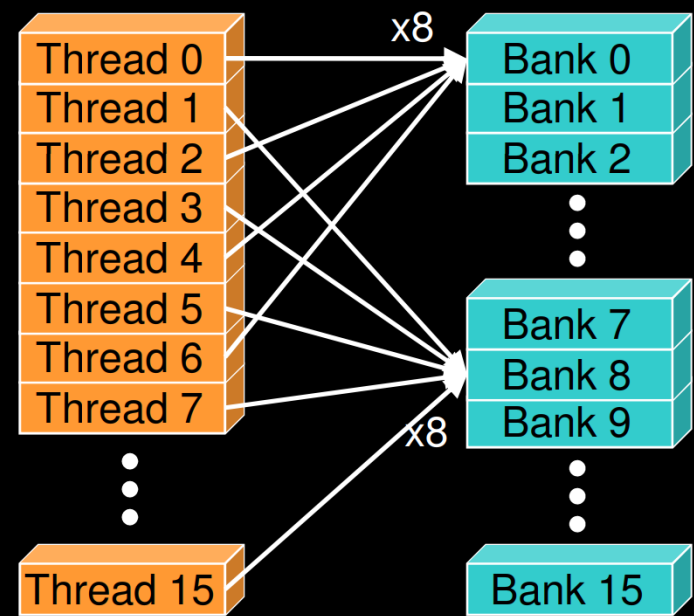


# SHARED MEMORY EXAMPLES

- **2-way Bank Conflicts**
  - Linear addressing stride == 2




- **8-way Bank Conflicts**
  - Linear addressing stride == 8



# PERF-OPT QUICK REFERENCE CARD

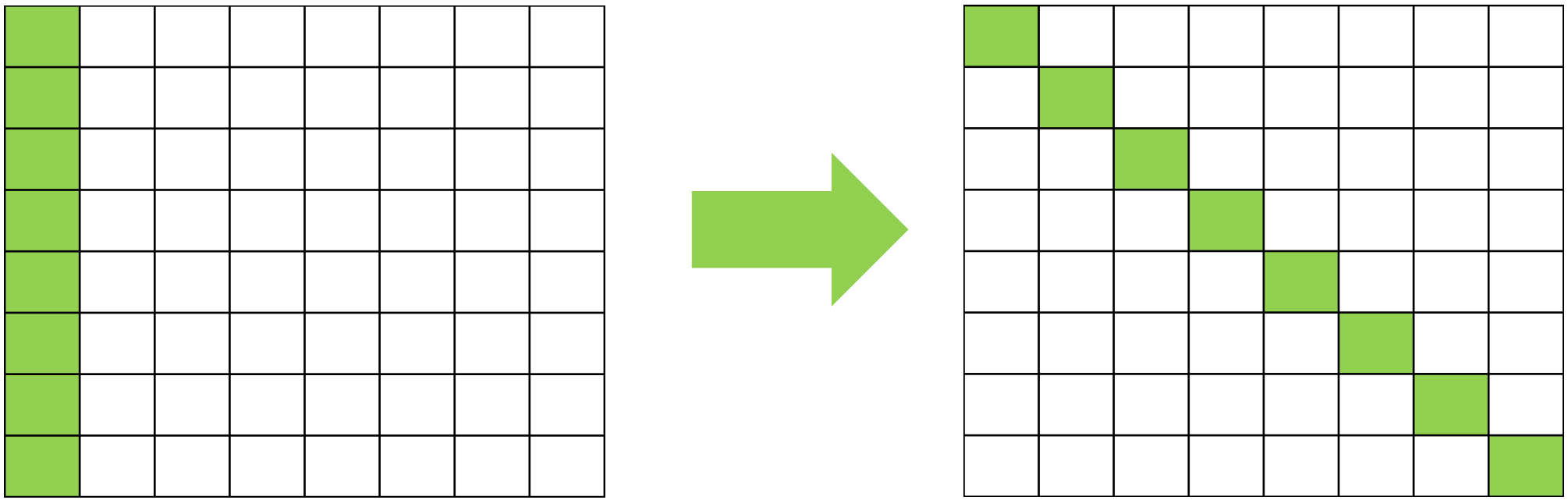
| Category: | Device Mem Bandwidth Bound - Shared Memory |
|---|---|
| Problem: | Too much data movement |
| Goal: | Reduce amount of data traffic to/from global mem |
| Indicators: | Higher than expected memory traffic to/from global memory<br>Low arithmetic intensity of the kernel |
| Strategy: | (Cooperatively) move data closer to SM:<br>• Shared Memory<br>• (or Registers)<br>• (or Constant Memory)<br>• (or Texture Cache) |

# PERF-OPT QUICK REFERENCE CARD

| Category: | Shared Mem Bandwidth Bound - Shared Memory |
|---|---|
| Problem: | Shared memory bandwidth bottleneck |
| Goal: | Reduce amount of data traffic to/from global mem |
| Indicators: | Shared memory loads or stores saturate |
| Strategy: | Reduce Bank Conflicts (insert padding)<br>Move data from shared memory into registers<br>Change data layout in shared memory |

# USING SHARED MEMORY WITHOUT CONFLICTS

# FIXING BANK CONFLICTS



# FIXING BANK CONFLICTS

```
// No bank-conflict transpose
// Same as transposeCoalesced except the first tile dimension is padded
// to avoid shared memory bank conflicts.
__global__ void transposeNoBankConflicts(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

```
Device : Tesla P100-PCIE-16GB
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
dimGrid: 32 32 1. dimBlock: 32 8 1
                  Routine         Bandwidth (GB/s)
                     copy              442.94
       shared memory copy              472.60
          naive transpose              111.13
      coalesced transpose              363.88
  conflict-free transpose              465.00
```

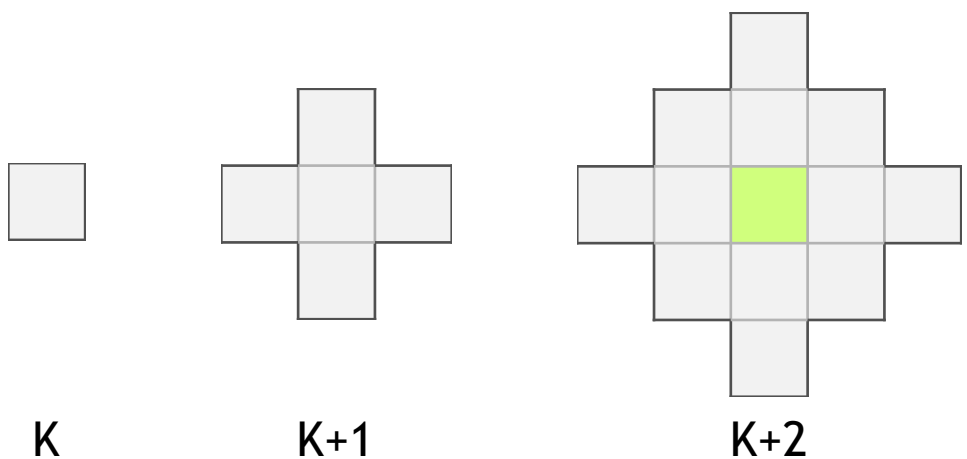
# ITERATION 3: KERNELS WITH INCREASED ARITHMETIC INTENSITY

# STENCIL OPERATIONS

## 2nd order vs 1st order

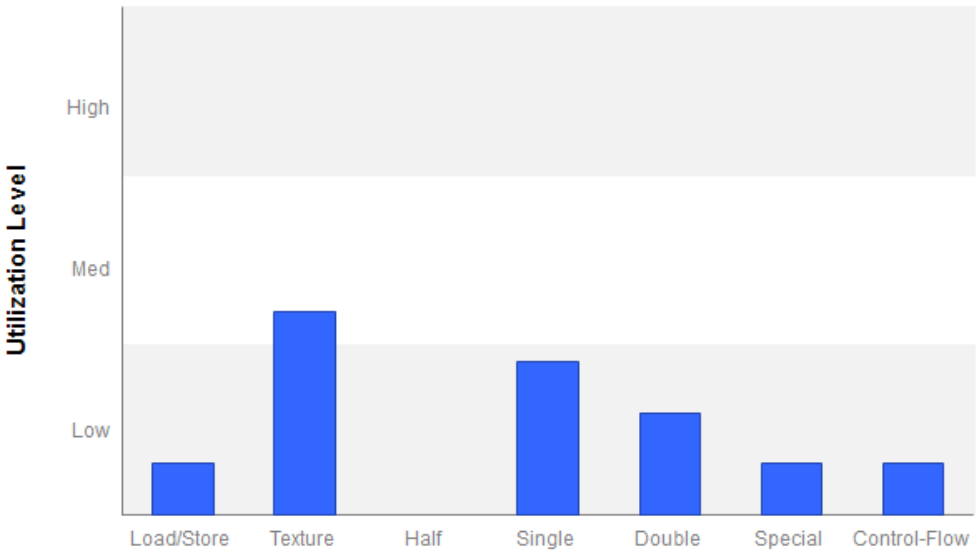
Performs 4x the FP operations

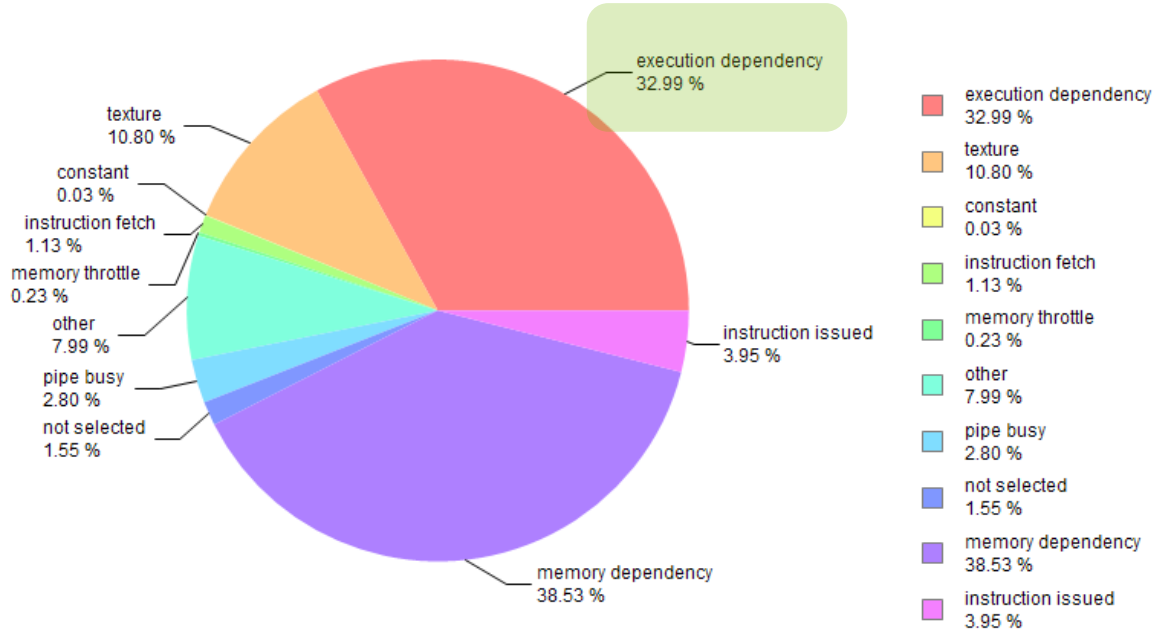
DRAM memory footprint is the same (assuming no over fetch)

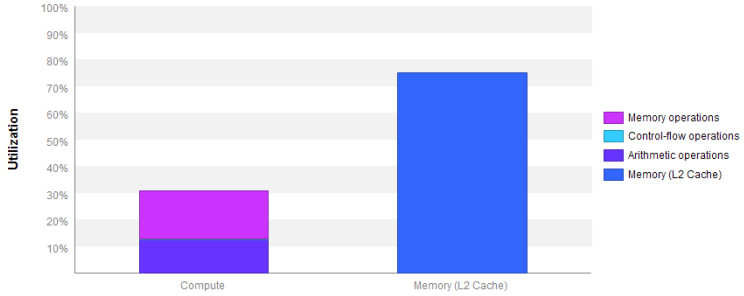


# FUNCTION UNIT UTILIZATION AND STALL REASONS

Execution Dependencies starting to become significant!

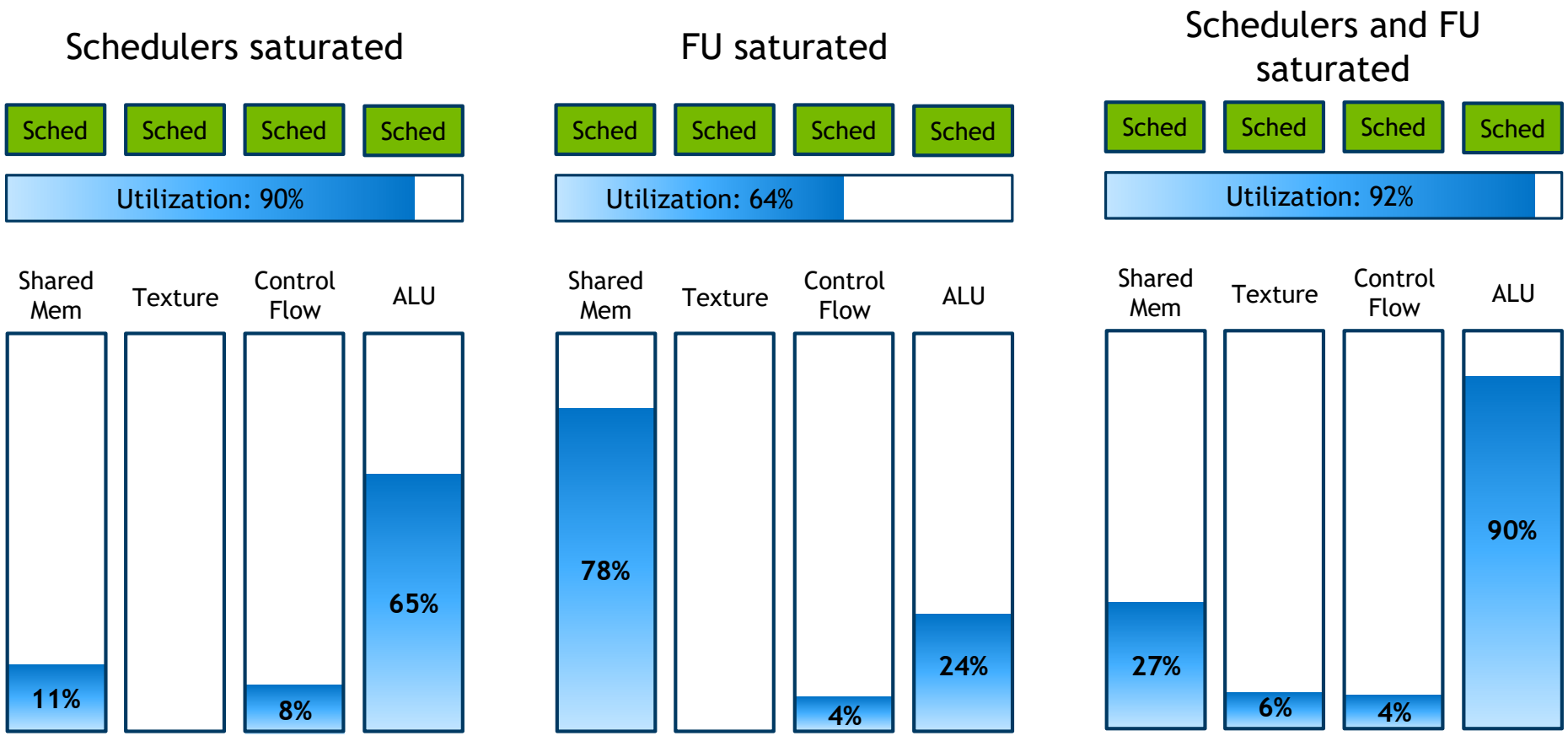



Functional units are not the bottlenecks in Stencils, even with higher order stencils!

# INSTRUCTION THROUGHPUT



# STALL REASONS: EXECUTION DEPENDENCY

```
a = b + c; // ADD
```

```
d = a + e; // ADD
```

```
a = b[i];  // LOAD
```

```
d = a + e; // ADD
```

Memory accesses may influence execution dependencies

- Global accesses create longer dependencies than shared accesses
- Read-only/texture dependencies are counted in Texture

Instruction level parallelism can reduce dependencies

```
a = b + c;  // Independent ADDs
d = e + f;
```

# ILP AND MEMORY ACCESSES

## No ILP

```
float a = 0.0f;
for( int i = 0 ; i < N ; ++i )
  a += logf(b[i]);
```

```
c = b[0]
a += logf(c)
c = b[1]
a += logf(c)
c = b[2]
a += logf(c)
c = b[3]
a += logf(c)
```

## 2-way ILP (with loop unrolling)

```
float a, a0 = 0.0f, a1 = 0.0f;
for( int i = 0 ; i < N ; i += 2 )
{
  a0 += logf(b[i]);
  a1 += logf(b[i+1]);
}
a = a0 + a1
```

```
c0 = b[0]
a0 += logf(c0)
c0 = b[2]
a0 += logf(c0)
...
a = a0 + a1
```

```
c1 = b[1]
a1 += logf(c1)
c1 = b[3]
a1 += logf(c1)
```

#pragma unroll is useful to extract ILP

Manually rewrite code if not a simple loop

# PERF-OPT QUICK REFERENCE CARD

| Category: | Bandwidth Bound - Register Caching |
|---|---|
| Problem: | Data is reused within threads and memory bw utilization is high |
| Goal: | Reduce amount of data traffic to/from global mem |
| Indicators: | High device memory usage, latency exposed<br>Data reuse within threads and small-ish working set<br>Low arithmetic intensity of the kernel |
| Strategy: | • Assign registers to cache data<br>• Avoid storing and reloading data (possibly by assigning work to threads differently)<br>• Avoid register spilling |

# PERF-OPT QUICK REFERENCE CARD

| Category: | Latency Bound - Instruction Level Parallelism |
|---|---|
| Problem: | Not enough independent work per thread |
| Goal: | Do more parallel work inside single threads |
| Indicators: | High execution dependency, increasing occupancy has no/little positive effect, still registers available |
| Strategy: | • Unroll loops (#pragma unroll)<br>• Refactor threads to compute n output values at the same time (code duplication) |

# PERF-OPT QUICK REFERENCE CARD

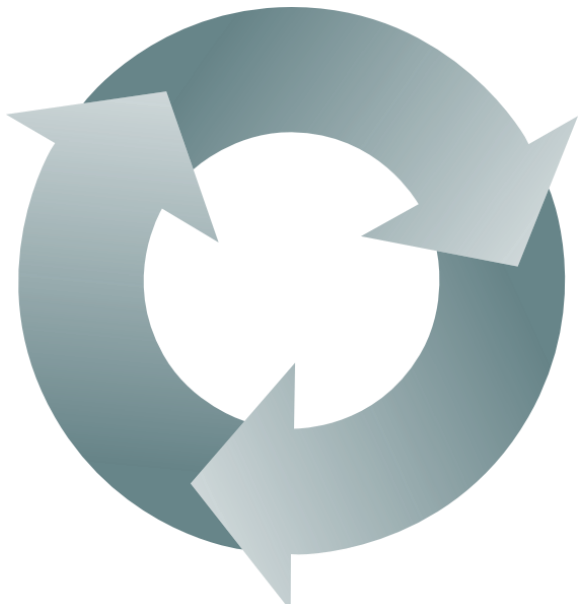
| Category: | Compute Bound - Algorithmic Changes |
|---|---|
| Problem: | GPU is computing as fast as possible |
| Goal: | Reduce computation if possible |
| Indicators: | Clearly compute bound problem, speedup only with less computation |
| Strategy: | • Pre-compute or store (intermediate) results<br>• Trade memory for compute time<br>• Use a computationally less expensive algorithm<br>• Possibly: run with low occupancy and high ILP |

# SUMMARY

# SUMMARY

## Performance Optimization is a Constant Learning Process

1. Know your application
2. Know your hardware
3. Know your tools
4. Know your process
   - Identify the Hotspot
   - Classify the Performance Limiter
   - Look for indicators
5. Make it so!



# REFERENCES

## CUDA Documentation

Best Practices: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

Kepler Tuning Guide: http://docs.nvidia.com/cuda/kepler-tuning-guide

Maxwell Tuning Guide: http://docs.nvidia.com/cuda/maxwell-tuning-guide

Pascal Tuning Guide: http://docs.nvidia.com/cuda/pascal-tuning-guide

## Parallel ForAll devblog

http://devblogs.nvidia.com/parallelforall/

## GTC Sessions:

http://on-demand-gtc.gputechconf.com


THANK YOU
NVIDIA