# Parallel Processing

## Unit 5



Dr. Ahmed Sallam

www.sallam.cf > (sallamah.weebly.com)

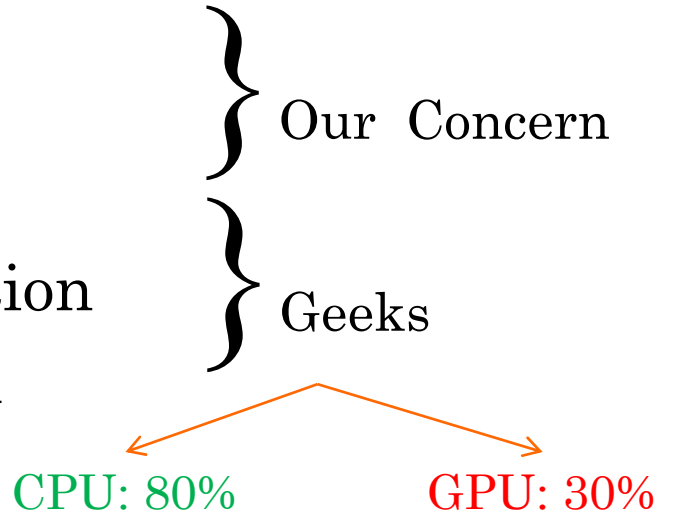
# QUIZ

Principals of Efficient GPU Programming

- ☐ Decrease arithmetic intensity
- ☑ Decrease time spent on memory operations
- ☑ Coalesce global memory accesses
- ☐ Move all data to shared memory
- ☐ Do fewer memory operations per thread
- ☑ Avoid thread Divergence

# Levels of Optimization

1. Picking good Algorithms
2. Basic principles for efficiency
3. Arch-specific detailed optimization
4. Instructions micro optimization




- Quiz

3 Vector registers – SSE, AVX

1 Use Odd-Even $O(n^2)$, Merge Sort O(nlogn), Or Heap sort(nlogn)

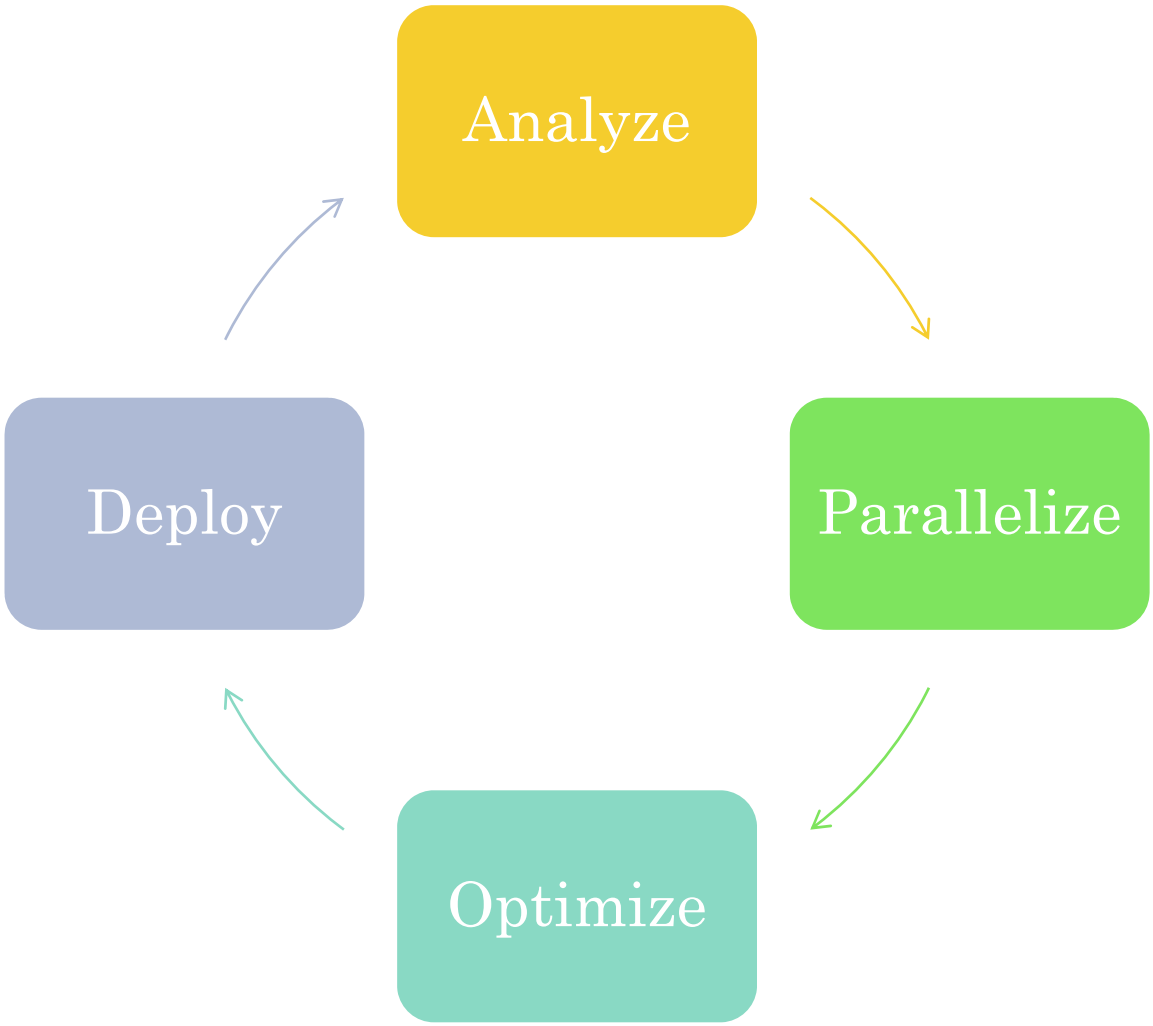
2 Write cache aware code (Iterate 2D array by column)

3 Block for L1 cache

4 Perform $x^{-½}$ by: 0x5f3759df - (x >> 1) (Quake3 game)

# APOD
## Systematic Optimization

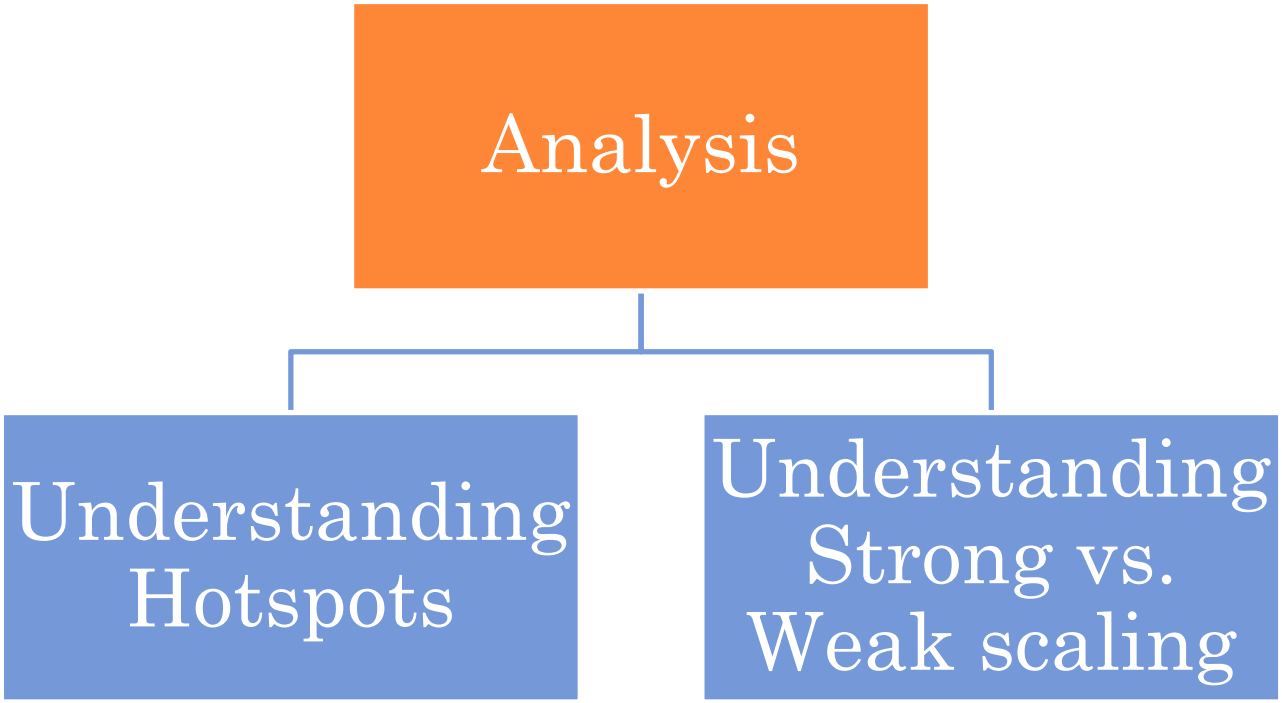
# APOD



# APOD (cont.)

- Analyze: Profile whole application
  - Where can it benefit?
  - By how much?
- Parallelize:
  - Pick an Approach
    - Complete algorithms: find Libraries or write by yourself (CUDA or OpenCl )
    - Small code: Directives (OpenMP for CPU), (OpenACC for GPU)
  - Pick an algorithm
- Optimization: Profile-driven Optimization
  - (Measure don't just guess)
- Deploy: Optimize to a limit
  - don't waste your time by more optimization.

# Analysis

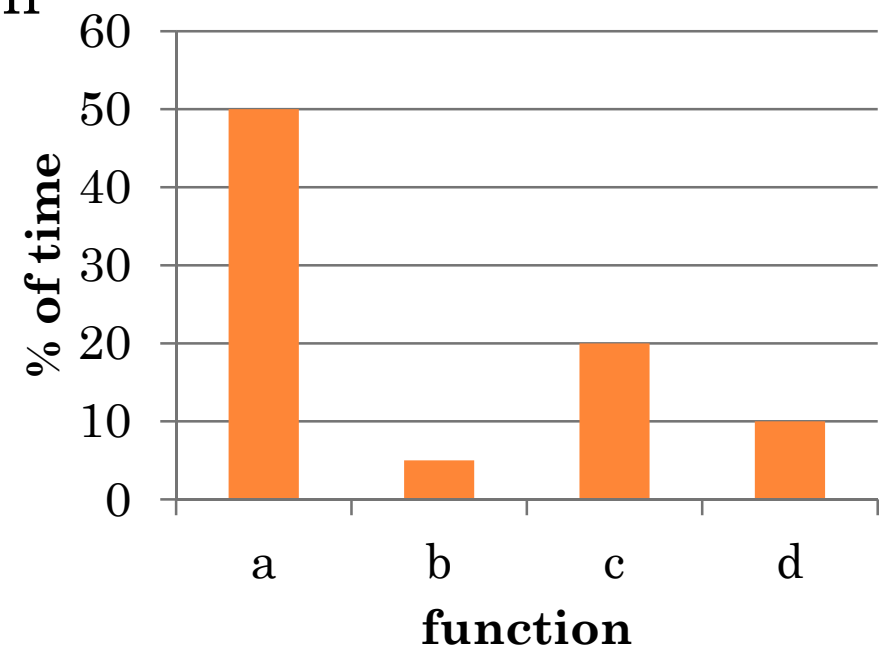

Analysis
Understanding Hotspots
Understanding Strong vs. Weak scaling

# Weak vs. Strong Scaling

- Serial program solves problem size A in time T
  - Ex: PageRank matrix multiplication in an hour
- Weak Scaling: run a larger problem, or more small one.
- Strong Scaling: run the same size problem more faster.

# Understanding Hotspots

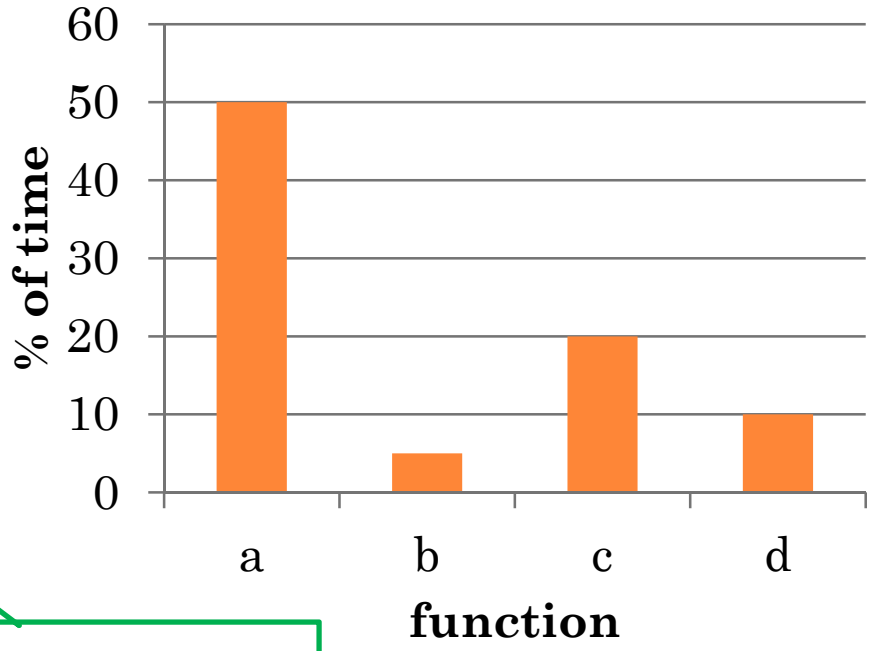
- Don't rely on intuition
- Run a profiler
  - Gprof
  - VTune
  - Very Sleepy




- Max speed up if you parallelize (Drop to zero):
  - Speedup by parallelize function (a): = 2x
  - Speedup by parallelize functions (a, c, d): = 5x

# Understanding Hotspots (cont.)

- Amdahl's Law
  - Total speedup from parallelization is limited by portion of time spent doing something to be parallelized.




- max speedup $= \frac{1}{s+\frac{p}{n}}$

  where `s` is serial portion and `p=(1-s)` is parallel portion, n is the number of threads

  $= \frac{1}{s} = \frac{1}{1-p}$

We drop this part to 0 because we have enough n

- Previous example:
  - p(a): 1/(1-50%) = 2x
  - p(a, c, d): 1/(1-80%) = 5x

# Parallelize

# MATRIX TRANSPOSE - SERIAL

$$\begin{array}{c|ccc} \frac{c}{r} & 0 & 1 & 2 \\ 0 & a & b & c \\ 1 & d & e & f \\ 2 & g & h & i \end{array} \rightarrow \begin{array}{ccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ [a & b & c & d & e & f & g & h & i] \end{array}$$

```
for (int r = 0; r < N; r++)
    for (int c = 0; c < N; c++)
        Out[r][c] = in [c][r];
```

$$\begin{array}{c|ccc} \frac{c}{r} & 0 & 1 & 2 \\ 0 & & & \\ 1 & & & \\ 2 & & & \end{array} \xrightarrow[c=0,r=0]{out[0]=in[0]} \begin{array}{c|ccc} \frac{c}{r} & 0 & 1 & 2 \\ 0 & a & & \\ 1 & & & \\ 2 & & & \end{array} \xrightarrow[c=0,r=1]{out[3]=in[1]} \begin{array}{c|ccc} \frac{c}{r} & 0 & 1 & 2 \\ 0 & a & & \\ 1 & b & & \\ 2 & & & \end{array} \xrightarrow[c=0,r=2]{out[6]=in[2]}$$

$$\begin{array}{c|ccc} \frac{c}{r} & 0 & 1 & 2 \\ 0 & a & & \\ 1 & b & & \\ 2 & c & & \end{array} \xrightarrow[c=1,r=0]{out[1]=in[3]} \begin{array}{c|ccc} \frac{c}{r} & 0 & 1 & 2 \\ 0 & a & d & \\ 1 & b & & \\ 2 & c & & \end{array} \xrightarrow[c=1,r=1]{out[4]=in[4]} \begin{array}{c|ccc} \frac{c}{r} & 0 & 1 & 2 \\ 0 & a & d & \\ 1 & b & e & \\ 2 & c & & \end{array} \ \ldots$$



<Forward

# Matrix Transpose – Parallel "Simple"

```
for (int r = 0; r < N; r++)
    for (int c = 0; c < N; c++)
        out[r + c*N] = in[c + r*N]; // out(r,c) = in(c,r)
```

- # blocks=1, # threads= 1024

```
int r = threadIdx.x; //row by thread

for (int c = 0; c < N; c++)
    out[r + c*N] = in[c + r*N]; // out(r,c) = in(c,r)
```

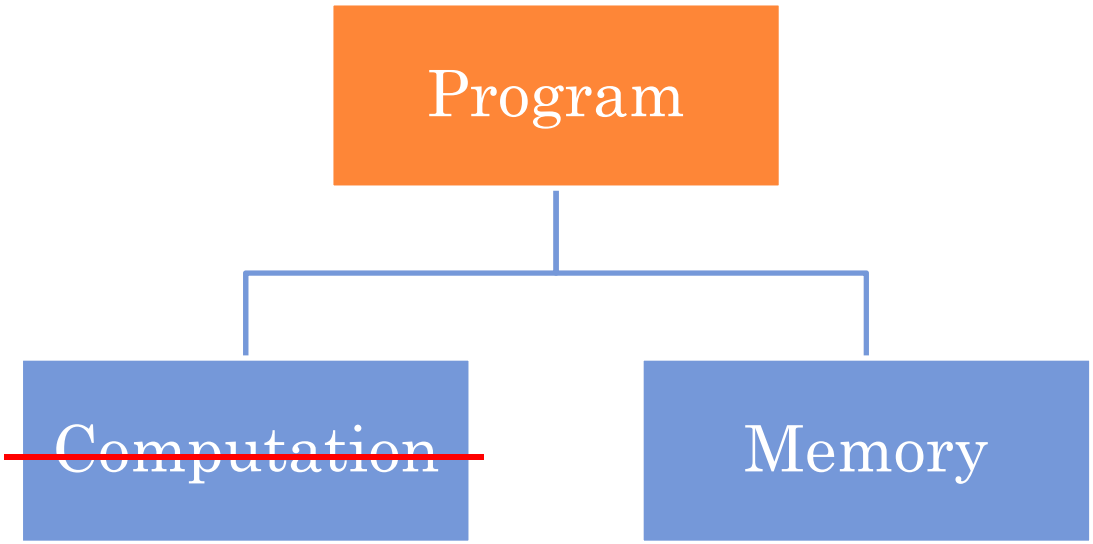
- # blocks=1024, # threads= 1024

```
int r = blockIdx.x; //row
int c = threadIdx.x; //column

out[r + c*N] = in[c + r*N]; // out(r,c) = in(c,r)
```

# Optimize

## Memory

# Optimization




Device Query ( Start )

# OPTIMIZATION (CONT.1)

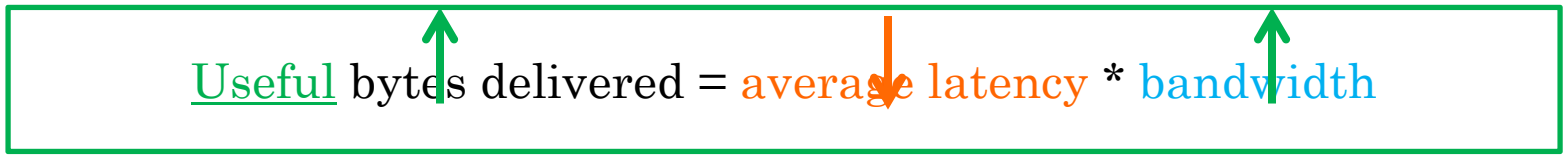
- A: Memory clock: 900 Mhz = $900 * 10^6$ clock per sec.
- B: Memory bus bandwidth: 64-bit = 8 byte per clock

Quiz: What is the theoretical peak bandwidth?

= A*B = $7200 * 10^6$ byte per sec.

- Simple parallel algorithm:
  - N= 1024
  - A: Time= 6.70794 ms
  - B: Memory $1024*1024 * 4_{(\text{sizeof(int)})} * 2_{(\text{read and write})} = 2^{23}$
  - Bandwidth= B/A = $1250 * 10^6$ byte per sec
- Performance= 1250/7200 = 17% <40 % Bad

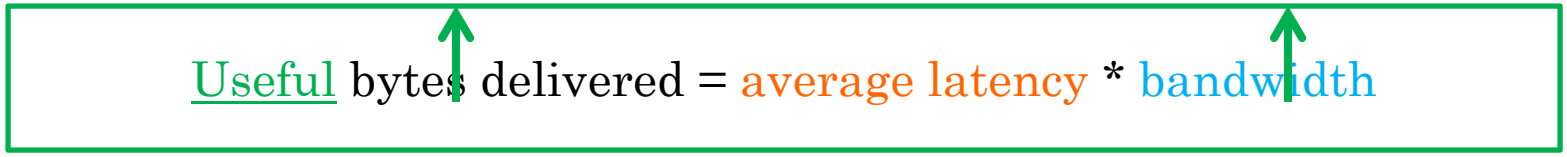
# LITTLE'S LAW



# OPTIMIZATION (CONT.3)

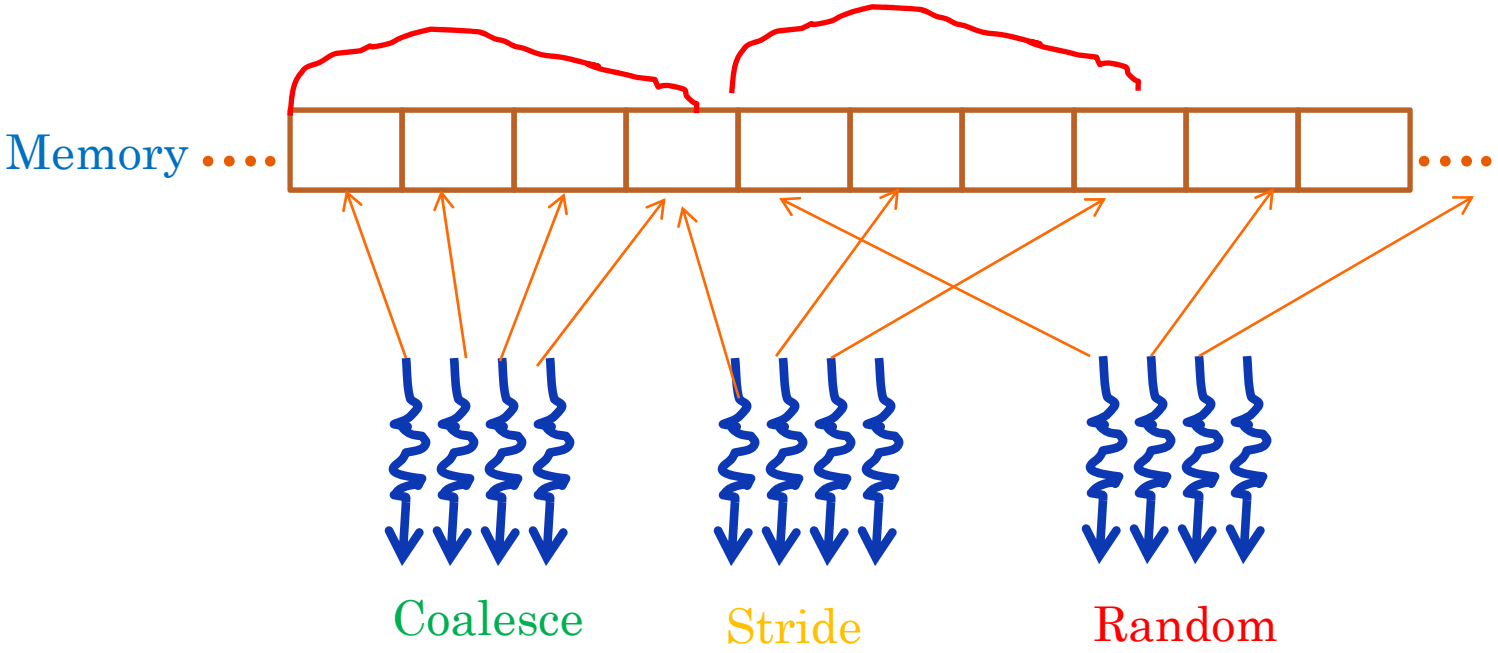
- nSight: Integrated debuggers and profilers, a complete development environment with many tools.
- Nvidia Visual Profiler (NVVP)

# LITTLE'S LAW




- Make sure the moved data are useful

# OPTIMIZATION (CONT.2)



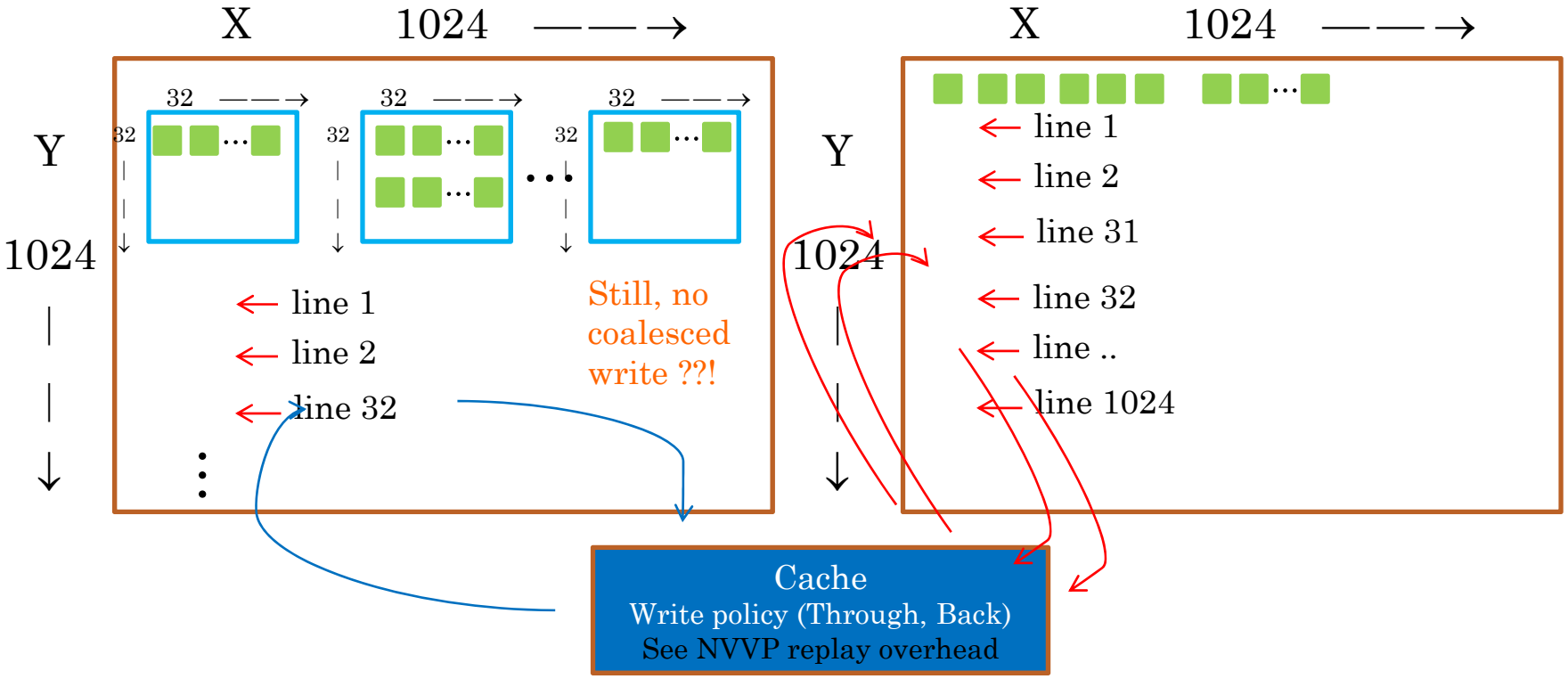

```
int r = blockIdx.x; //row
int c = threadIdx.x; //column

out[r + c*N] = in[c + r*N]; // out(r,c) = in(c,r)
```

(N step) (1 step)

Back>

# Matrix Transpose – Parallel "Tiled" [Coalesced]

- # blocks=32*32, # threads= 32*32




```
int r = blockIdx.y * K + threadIdx.y;
int c = blockIdx.x * K + threadIdx.x;

out[r + c*N] = in[c + r*N]; // out(r,c) = in(c,r)
```

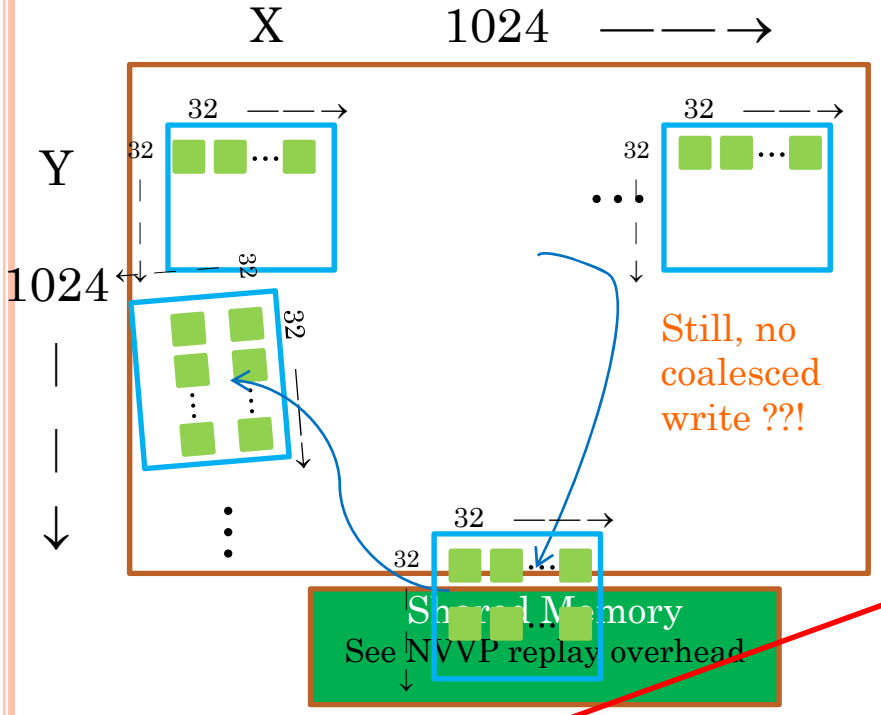
# LITTLE'S LAW

Useful bytes delivered = average latency * bandwidth

- DRAM latency = 100's clock cycle
- Move to Shared memory

# Matrix Transpose – Parallel "Tiled Shared Memory"

- # blocks=32*32, # threads= 32*32




```
int x = threadIdx.x, y = threadIdx.y;

int in_r = blockIdx.y * K + y,
    out_r = blockIdx.x * K + y;
int in_c = blockIdx.x * K + x,
    out_c = blockIdx.y * K + x;

__shared__ float tile[K][K];

// coalesced read from global mem,
// write into shared mem:
tile[y][x] = in[in_c + in_r*N];
__syncthreads();
//TRANSPOSED read from shared mem,
//coalesced write to global mem:
out[out_c + out_r*N] = tile[x][y];
```

Overhead

32 * 4 > warp limit
(See also NVVP replay overhead)

# OCCUPANCY

- Each SM has a limited number of: Device Query ( Start )
  - Blocks (currently 8)
  - Threads (currently 1536 to 2048)
  - Registers (currently 64K)
  - Shared memory (Currently 16k to 48k )
- All these factors are related:
  - Example 1:
    - If my GPU has 48 k shared memory
    - And my kernel takes 16k per block
    - Then my SM can only run 3 blocks at a time
  - Example 2:
    - If my SM can run 1536
    - And my kernel runs 1024 thread per block
    - Then my kernel can only run 1 block at a time
- If I run only 1024 thread then I waste 1536 – 1024 which considered bad Occupancy

# OCCUPANCY (CONT.1)

- Check our tiled_shared kernel occupancy

Start

- How many blocks to run on an SM?

| | My Kernel | My system Limit | Possible blocks |
|---|---|---|---|
| | | | |
| Registers/ Thread | | | |
| Shared memory | | | |
| Blocks/SM | | | |

# LITTLE'S LAW

Useful bytes delivered = average latency * bandwidth

- _syncthreads() cause overhead and decrease the latency because a thread may wait long time for other threads to finish.
- Decrease the number of threads per block

# OCCUPANCY (CONT.2)

- Shared memory bank conflict: shared memory organized in banks.

```
int x = threadIdx.x, y = threadIdx.y;

int in_r = blockIdx.y * K + y, out_r = blockIdx.x * K + y;
int in_c = blockIdx.x * K + x, out_c = blockIdx.y * K + x;

__shared__ float tile[K][K + 1];

// coalesced read from global mem,  write into shared mem:
tile[y][x] = in[in_c + in_r*N];
__syncthreads();
// TRANSPOSED read from shared mem, coalesced write to global mem:
out[out_c + out_r*N] = tile[x][y];
```

# Optimize

## Computation

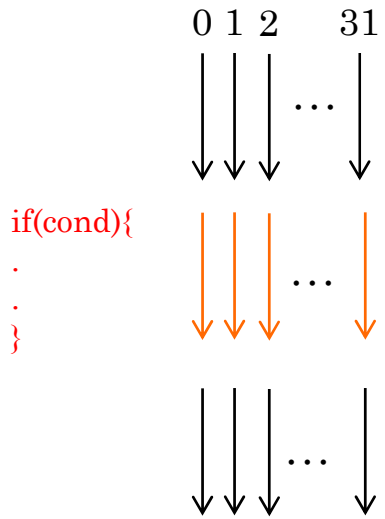
# Computation Optimization

- Minimize time spent at barriers
- Minimize thread divergence
- Math Optimization
- CPU-GPU interaction optimization
- Streams

# MINIMIZE THREAD DIVERGENCE

- Warp: set of threads execute the same instruction at a time.
- SIMD: Single Instruction Multiple Data
  - CPU SSE / AVX  victor registers
- SIMT: Single Instruction Multiple Thread

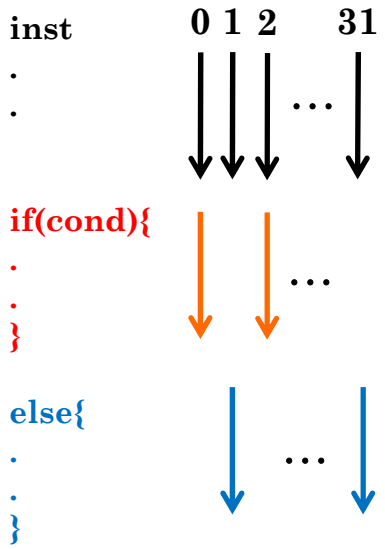
# THREADS WARP



# Branch Divergence

- What is the maximum branch divergence penalty for a CUDA thread block with 1024 threads?

  32x slowdown




2-way branch divergence

# QUIZ

What will be the slowdown for each of the following expressions in switch statement?

32
```
__global__ void foo(){  switch (threadIdx.x%32)  case(0…31)}
  kernel <<1024, 1>> ();
```

32
```
__global__ void foo(){  switch (threadIdx.x%64)  case(0…63)}
 kernel <<1024, 1>> ();
```

1
```
__global__ void foo(){  switch (threadIdx.y%32)  case(0…31)}
```
Next>

```
kernel <<64*16, 1>> ();
```

2
```
__global__ void foo(){switch (threadIdx.y%16)case(0…31)}
kernel <<16*16, 1>> ();
```
Next>

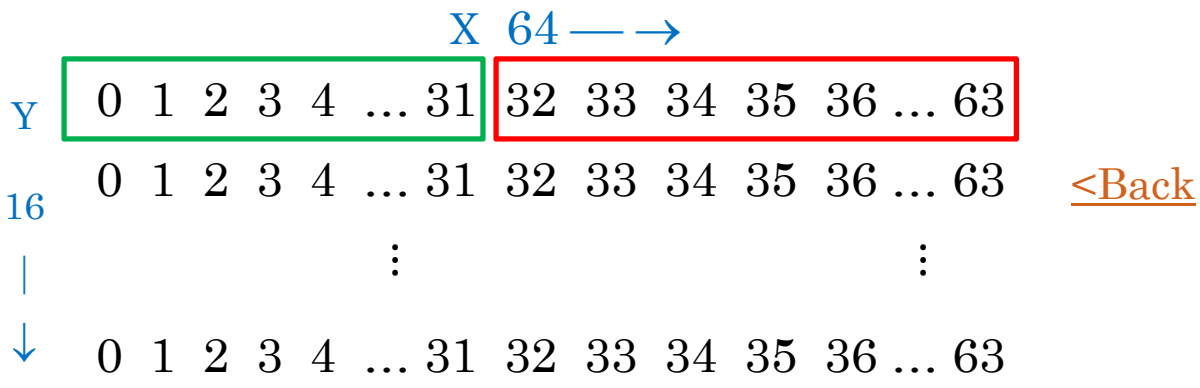
# QUIZ (CONT.1)

- CUDA assigns thread ID as following:

  x varies fast

  y varies slower

  z varies slowest




<Back

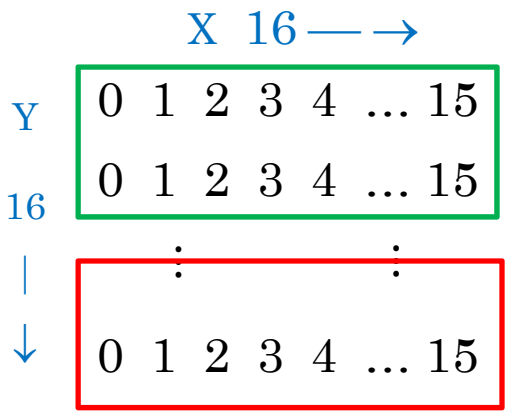
# QUIZ (CONT.2)

- CUDA assigns thread ID as following:

  x varies fast

  y varies slower

  z varies slowest




<Back

# Loop Divergence

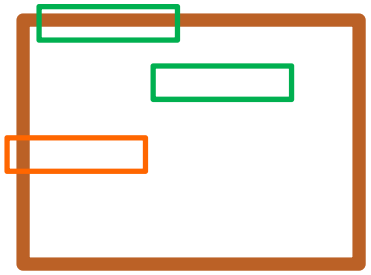
```
__global__ void foo(){
        for(int i=0; i < threadIdx.x%32; i++) //A
                bar();

        for(int i=0; i < threadIdx.x%32; i++) //B
                bar();
}

kernel <<1024, 1>> ();
```

- Which loop is faster?
  - 2nd
- How much fast?
  - 2 times

# Real world Divergence Example

- Operating on 1024 x 1024 image special handling of pixels on the boundary

```
__global__ perPixelKernel(float3 * image){
          if(threadIdx.x==0 || threadIdx.x==1024 ||
             threadIdx.y==0 || threadIdx.y==1024){
                    operation_on_boundries();
          }
          else{
                    operation_on_pixels();
          }

}
```



- What is the maximum branch divergence of any warp?
  - 2 way
- How many total threads are in diverged warps- warps with at least 2-way divergence?
  - 2*1022

# OPTIMIZE BRANCH DIVERGENCE

- Avoid branching code
  - Lots of if and switch statements
  - Try to restructure
- Beware large imbalance  in thread workloads
  - Ex. When some threads perform the loop more than other threads

# Computation Optimization

- Minimize time spent at barriers
- Minimize thread divergence
- Math Optimization
- CPU-GPU interaction optimization
- Streams

# Math Optimization

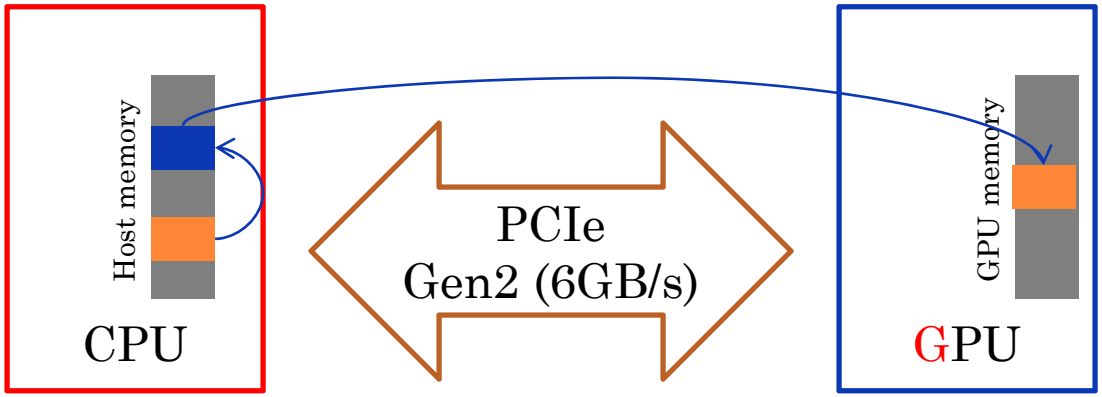
- Use double precision when only needed
- 64-bit math slower than 32-bit math
  - float a=b+2.5 > float a=b+2.5**f**
- Use intrinsics when possible
  - __sin(), sqrt() (2-3 bits less precision but faster) math.h

# Computation Optimization

- Minimize time spent at barriers
- Minimize thread divergence
- Math Optimization
- CPU-GPU interaction optimization
- Streams

# HOST-GPU INTERACTION

- Swap-out problem



# HOST-GPU INTERACTION OPTIMIZATION

- Use pinned host memory (page-locked)
  - cudaMallocHost, cudaHostRegister instead of Malloc
- If you are using Pinned memory:
  - Use cudaMemcpyAsync instead of cudaMemcpy
  - This help to free the CPU
  - But you need to define stream

# Computation Optimization

- Minimize time spent at barriers
- Minimize thread divergence
- Math Optimization
- CPU-GPU interaction optimization
- Streams

# STREAMS

- Stream: sequence of operations that execute in order.
  - (Memory transfers, kernels)

- Ex.

```
cudaMemcpyAsync(..);
A<<..>>;
cudaMemcpyAsync(..);
B<<..>>;
```

✘

Default steam (s0)

```
cudaMemcpyAsync(.., s1);
A<<>>(..,s2);
cudaMemcpyAsync(.., s3);
B<<>>(.., s4);
```

✓

```
cuda stream_t s1;
cuda streamCreate(s1);
..
cuda streamDestroy(s1);
```

# Quiz

- If all operations take exactly 1 second, how long to complete the following:

```
cudaStream_t s1, s2;
cudaStreamCreate(&s1); cudaStreamCreate(&s2);

cudaMemcpy(&d_arr, &h_arr, numbytes, cudaH2D);
A<<<1, 128>>>(d_arr);
cudaMemcpy(&h_arr, &d_arr, numbytes, cudaD2H);

cudaMemcpyAsync(&d_arr, &h_arr, numbytes, cudaH2D, s1);
A<<<1, 128, s1>>>(d_arr);
cudaMemcpyAsync(&h_arr, &d_arr, numbytes, cudaD2H, s1);

cudaMemcpyAsync(&d_arr1, &h_arr1, numbytes, cudaH2D, s1);
A<<<1, 128, s1>>>(d_arr1);
cudaMemcpyAsync(&h_arr1, &d_arr1, numbytes, cudaD2H, s1);
cudaMemcpyAsync(&d_arr2, &h_arr2, numbytes, cudaH2D, s2);
B<<<1, 192, s2>>>(d_arr2);
cudaMemcpyAsync(&h_arr2, &d_arr2, numbytes, cudaD2H, s2);

cudaMemcpyAsync(&d_arr1, &h_arr1, numbytes, cudaH2D, s1);
cudaMemcpyAsync(&d_arr2, &h_arr2, numbytes, cudaH2D, s2);
A<<<1, 128, s1>>>(d_arr1);
B<<<1, 192, s2>>>(d_arr2);
cudaMemcpyAsync(&h_arr1, &d_arr1, numbytes, cudaD2H, s1);
cudaMemcpyAsync(&h_arr2, &d_arr2, numbytes, cudaD2H, s2);
```

3

3

3

3

# QUIZ [CONT.]

- If all operations take exactly 1 second, how long to complete the following:

```
cudaStream_t s1, s2;
cudaStreamCreate(&s1); cudaStreamCreate(&s2);
```

1

```
cudaMemcpyAsync(&d_arr1, &h_arr1, numbytes, cudaH2D, s1);
A<<<1, 128, s2>>>(d_arr2);
```

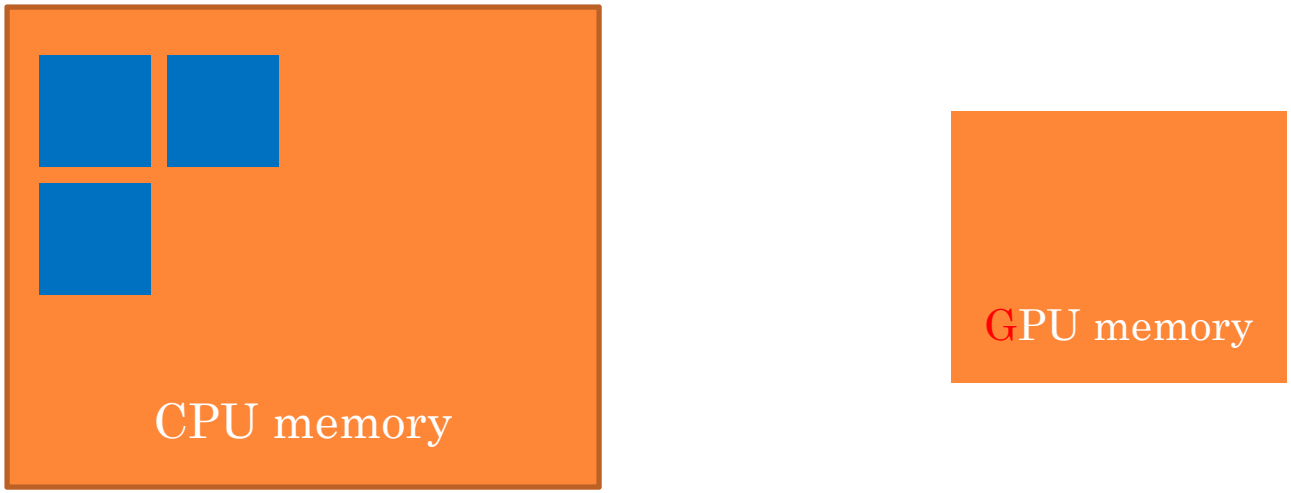
1

```
cudaMemcpyAsync(&d_arr1, &h_arr1, numbytes, cudaH2D, s1);
A<<<1, 128, s2>>>(d_arr1);
```

With
a wrong values

# STREAMS (CONT.)

- Old GPUs big amount of data to chunks
  - Too much overhead to transfer data back and forward

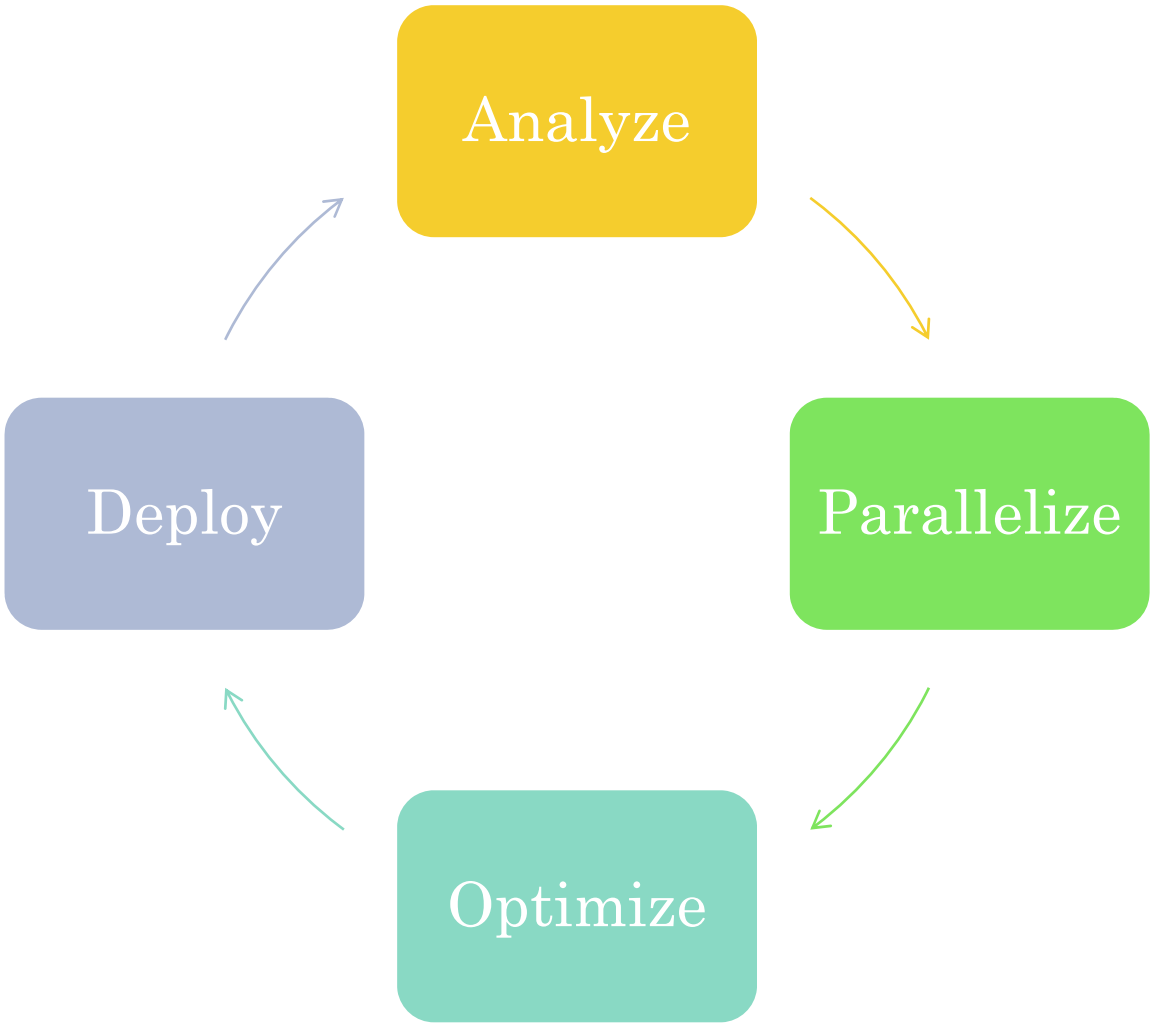



- Streams good with big amount of data
  - Run chunks and kernels in asynchronous manner
  - Overlap memory and compute
  - Fill GPU with smaller kernels

# Computation Optimization

- Minimize time spent at barriers
- Minimize thread divergence
- Math Optimization
- CPU-GPU interaction optimization
- Streams

# APOD



# PROBLEM SET

- Optimize the Histogram program in lecture 3
  - Use local bins instead of atomicAdd