Basic GPU Performance

CS378 – Spring 2015

Sreepathi Pai

Outline

- System Performance
- GPU Occupancy
- Data Layout and Work Distribution
- Static Scheduling of Work

System Performance

- GPU + CPU forms a heterogeneous system
  - “A system where programmer must choose where to perform a computation” (definition-in-progress)
- Parallel execution is possible
  - CPU and GPU can be working on work independently in parallel
  - In fact, GPU allows data transfers in parallel to GPU execution
- Consider distributing work so that all execution units (CPU and GPU) are fully occupied
- Not easy to do manually, but no automatic solution widely accepted yet

Measurement Pitfalls

Keep in mind:

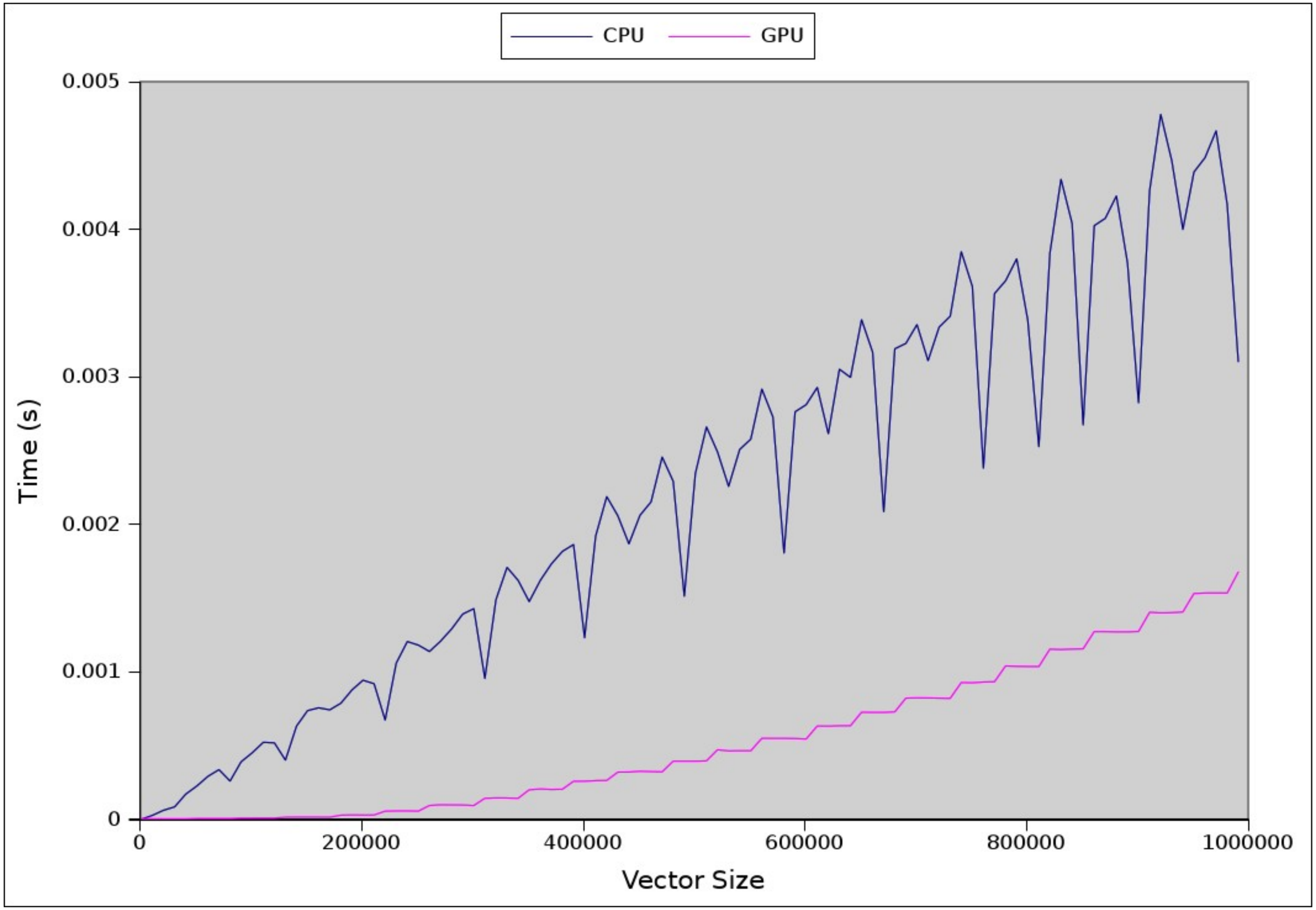
- A GPU program is a *parallel* CPU program
  - i.e. GPU code sometimes runs on a separate thread
- A CPU + GPU system is a distributed system
  - i.e. clocks are unsynchronized
  - especially across GPU cores
- Use *timelines* not *intervals* to reason about performance
  - timelines capture overlap
  - timelines illustrate critical path
  - NVIDIA Profiler provides timelines

How not to time a GPU kernel

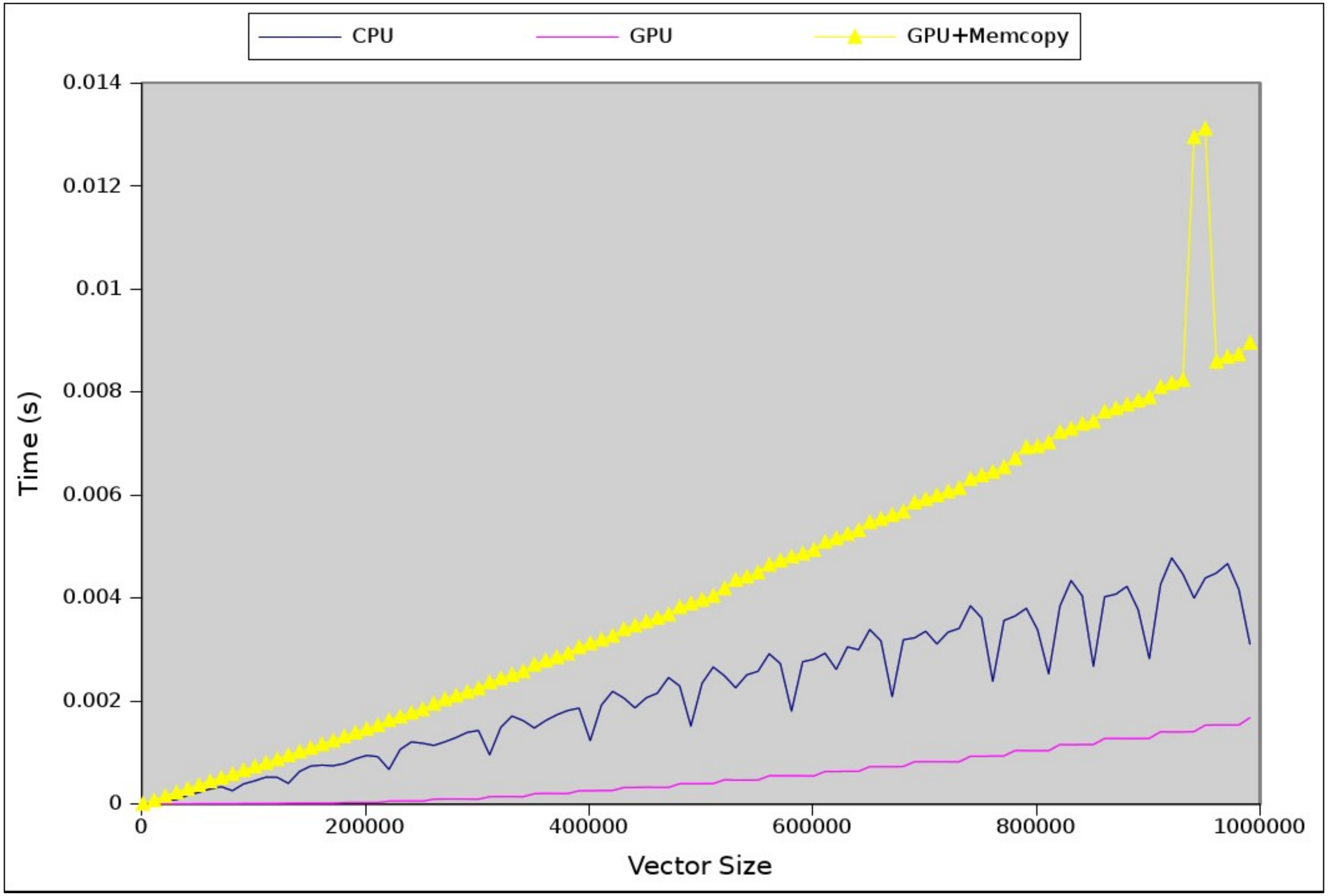
```
struct stopwatch va;

clock_start (&va ) ;
vector_add_1 <<<14∗8, 384>>>(ga , gb , gc , N) ;
clockstop (&va ) ;

printf (TIMEFMT ”s \n” , va.elapsed.tv_sec , va.elapsed.tv_nsec ) ;
```

- Output is approx. 40μs on my machine
- NVIDIA Compute Profiler:
  gputime=[14078.336] (μs)

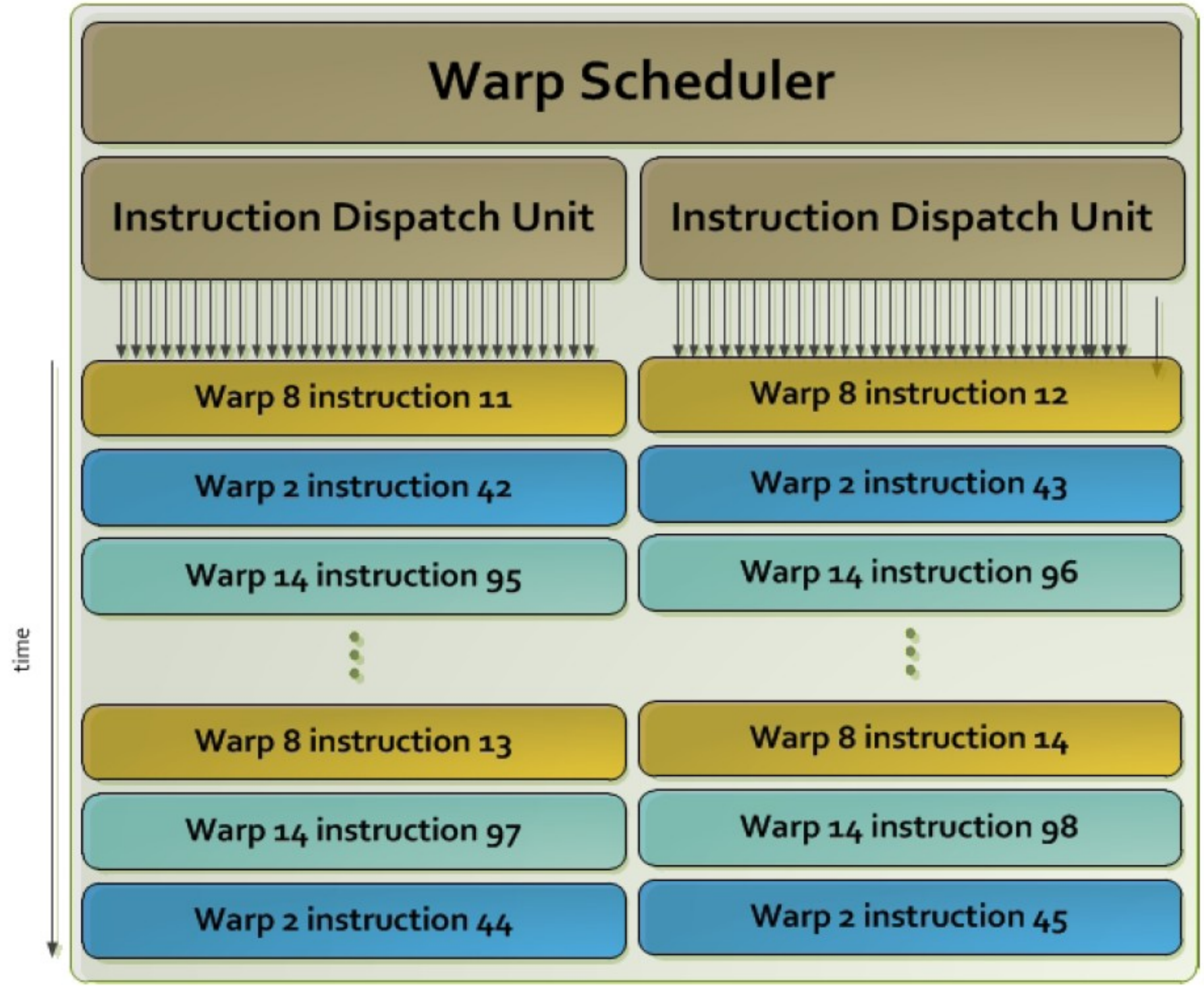
Vector-Addition



Vector Addition + Transfer Time



GPU Occupancy



GPU Occupancy – contd.

- GPUs divide resources among threads to enable hardware multithreading
- The number of *concurrent* threads is determined by the resource that is exhausted first
- Occupancy is the ratio of running concurrent threads to the maximum number of SM threads
- Residency is the number of thread blocks that can run concurrently on the SM
- NVIDIA provides an occupancy calculator that calculates this number for different GPUs

Resource	Available	Maximum
Threads	2048	1024/block
Shared Memory	48K (max)	48K/block
Registers	65536	255/thread
Thread Blocks	16	16/SM

Resources are per SM on NVIDIA Kepler

```
kernel<<<2048, 32>>>()
```

threads/block: 32
registers: 100/thread -> 3200/block
shared mem: 1K/block

residency: 16, exceeds maximum thread blocks
occupancy: 16*32/2048 = 25%

```
kernel<<<2048, 32>>>()
```

threads/block: 32
registers: 160/thread -> 5120/block
shared mem: 1K/block

residency: 12, exceeds maximum registers
occupancy: 12*32/2048 = 18%

Should occupancy be maximized?

- NVIDIA Manual – roughly, yes
- But:

Multiplication of two large matrices, single precision (SGEMM):

	CUBLAS 1.1	CUBLAS 2.0	
Threads per block	512	64	8x smaller thread blocks
Occupancy (G80)	67%	33%	2x lower occupancy
Performance (G80)	128 Gflop/s	204 Gflop/s	1.6x higher performance

Batch of 1024-point complex-to-complex FFTs, single precision:

	CUFFT 2.2	CUFFT 2.3	
Threads per block	256	64	4x smaller thread blocks
Occupancy (G80)	33%	17%	2x lower occupancy
Performance (G80)	45 Gflop/s	93 Gflop/s	2x higher performance

Volkov, V., “Better Performance at Lower Occupancy”, GTC 2010

Volkov's Summary

- Do more parallel work per thread to hide latency with fewer threads (i.e. increase ILP)
- Use more registers per thread to access slower shared memory less
- Both may be accomplished by computing multiple outputs per thread

  [Note that Volkov underutilizes threads, but maxes out registers!]

Data Layout

```
struct pt {
  int x;
  int y;
};

__global__
void aos_kernel(int n_pts, struct pt *p) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int nthreads = blockDim.x * gridDim.x;

  for(int i = tid; i < n_pts; i += nthreads) {
    p[i].x = i;
    p[i].y = i * 10;
  }
}


In main():

struct pt *p;
cudaMalloc(&p, ...)
```

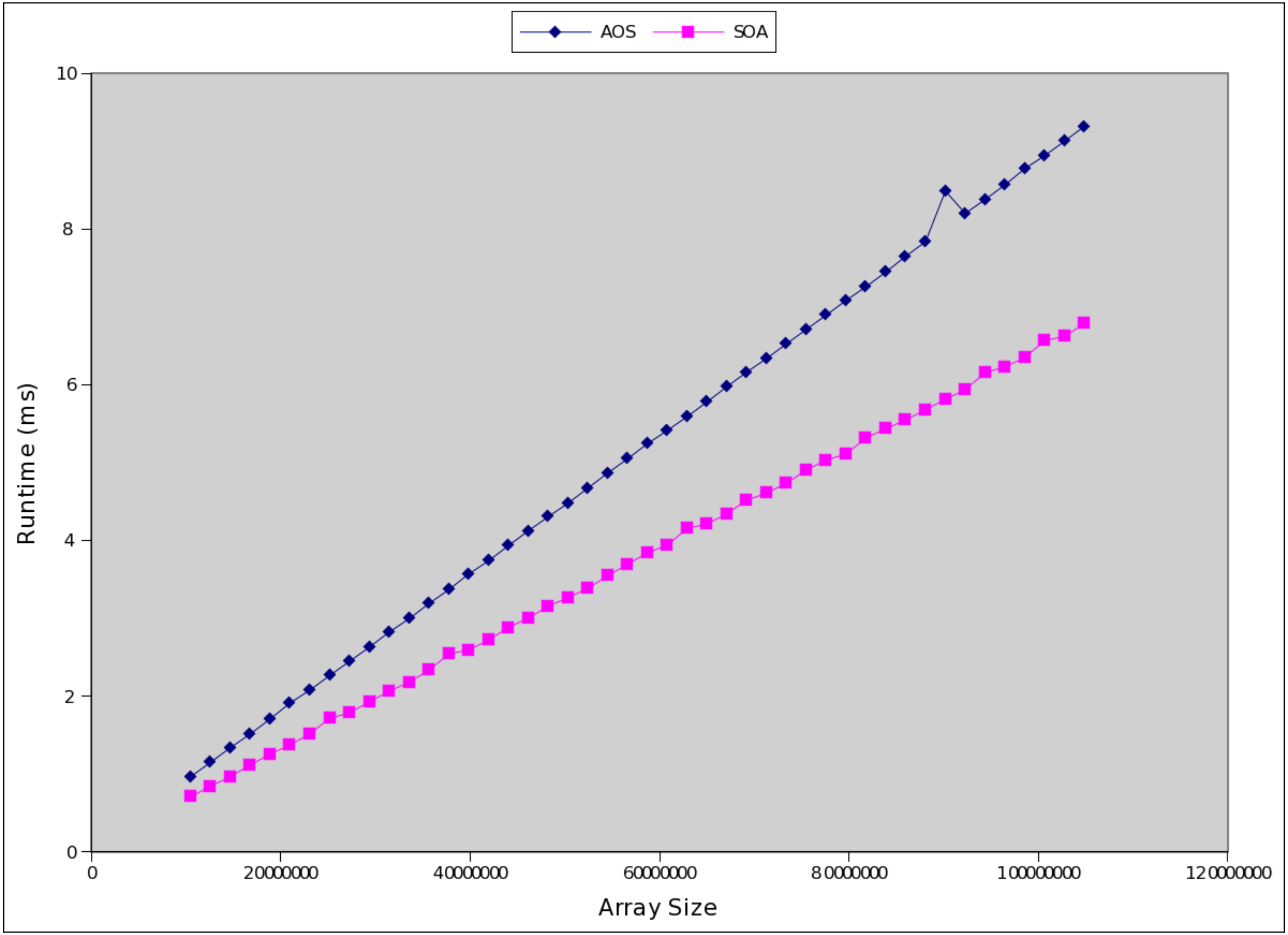
Array of Structure (AoS)

```
struct pt {
  int *x;
  int *y;
};

__global__
void soa_kernel(int n_pts, struct pt p) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int nthreads = blockDim.x * gridDim.x;

  for(int i = tid; i < n_pts; i += nthreads) {
    p.x[i] = i;
    p.y[i] = i * 10;
  }
}


In main():

struct pt p;
cudaMalloc(&p.x, ...)
cudaMalloc(&p.y, ...)
```

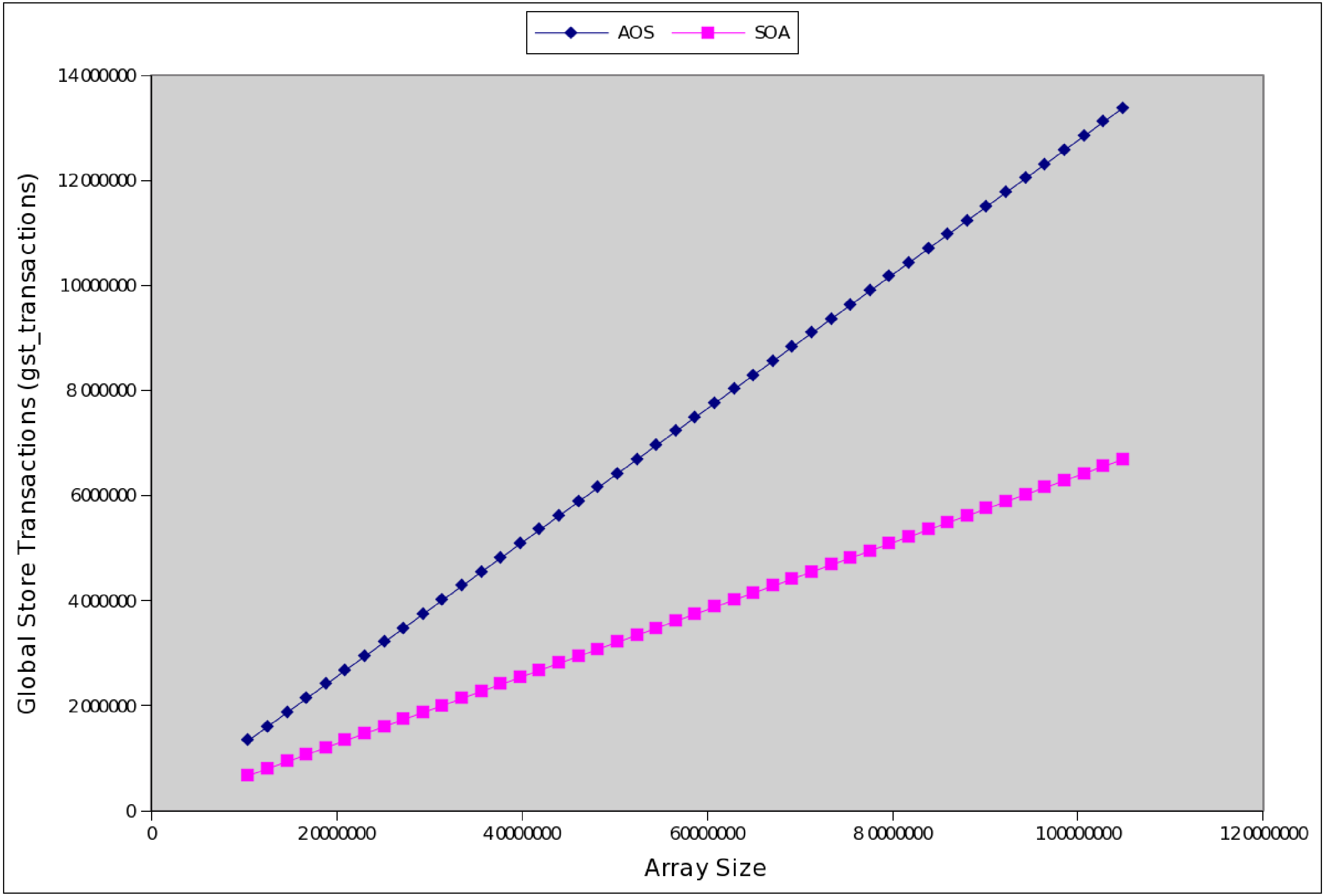
Structure of Arrays (SoA)

Which, if any, is faster?

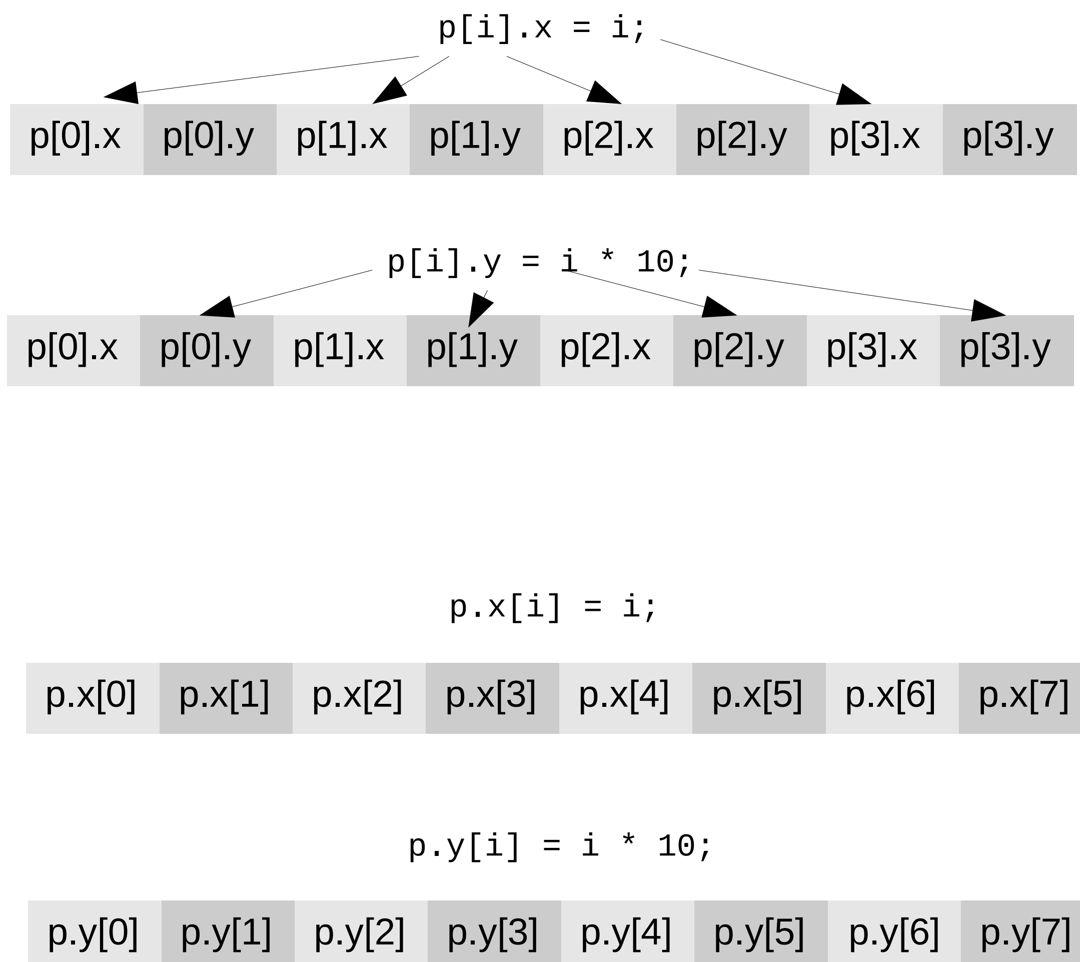
SoA vs AoS Results



Why?


AOS
SOA
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
Global Store Transactions (gst_transactions)
0
20000000
40000000
60000000
80000000
100000000
120000000
Array Size

AoS vs SoA memory layout



Assigning Work to Threads

```
start = tid * blksize;
end = start + blksize;

for(i = start; i < N && i < end; i++)
    a[i] = b[i] + c[i]
```

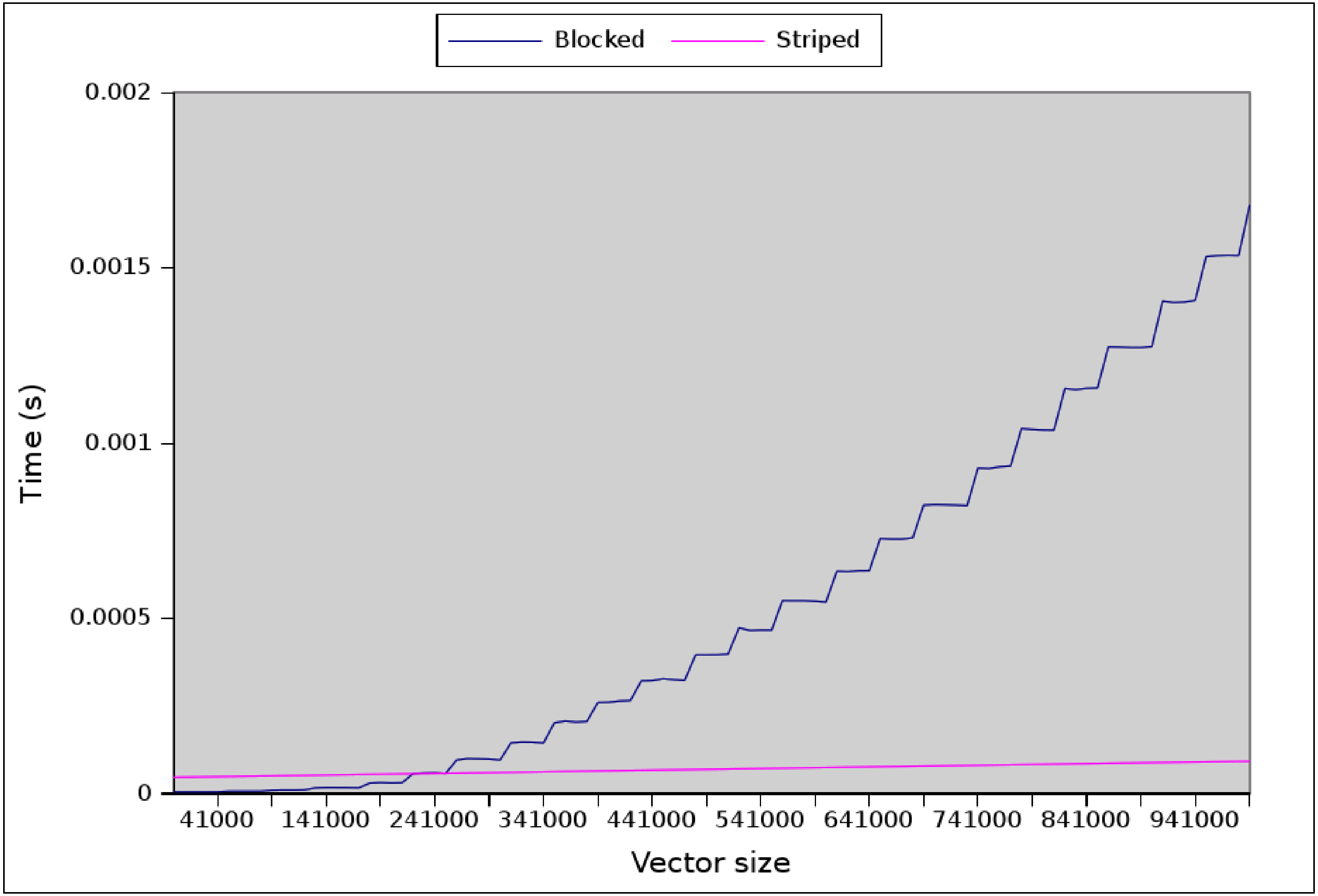
Blocked

```
start = tid;


for(i = start; i < N; i+=nthreads)
    a[i] = b[i] + c[i]
```

Interleaved/Striped

Which, if any, is faster?

Blocking v/s Striped



Exploiting Spatial Locality (1) Texture Cache

- Textures are 2-D images that are “wrapped” around 3-D models
- Exhibit 2-D locality, so textures have a separate cache
- GPU contains a texture fetch unit that non-graphics programs can also use
  - Step 1: map arrays to textures
  - Step 2: replace array reads by tex1Dfetch(), tex2Dfetch()
- Catch: Only read-only data can be cached
  - you can write to the array, but it may not become visible through the texture in the same kernel call
- Easiest way to use textures:
  - const __restrict__ *

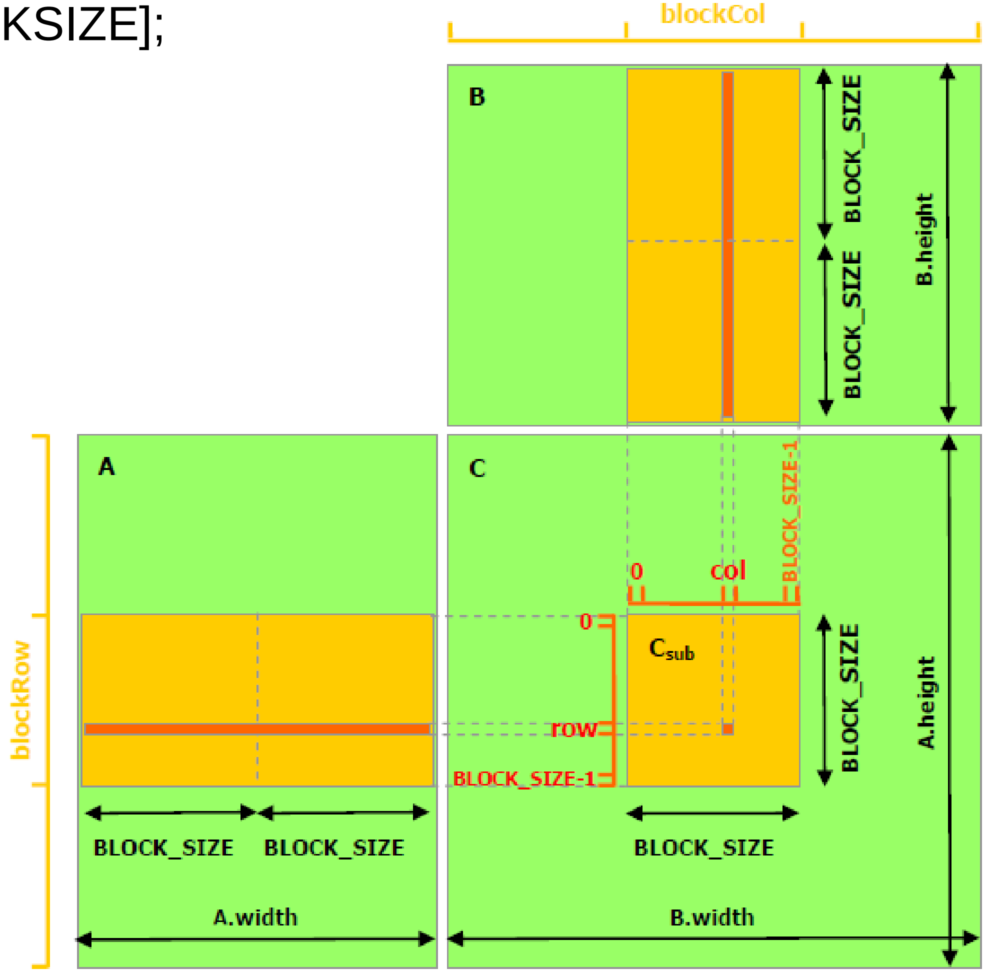
Exploiting Spatial Locality (2) Shared Memory

- “Shared Memory” is on-chip software-managed cache, also known as a scratchpad
- 48K maximum size
- Partitioned among thread blocks
- __shared__ qualifier places items in shared memory
- Can be used for communicating between threads of the same thread block

```
__shared__ int x;

if(threadIdx.x == 0)
        x = 1;

__syncthreads(); //required!

printf(“%d\n”, x);
```

Using Shared Memory (SGEMM)

```
__shared__ float c_sub[BLOCKSIZE][BLOCKSIZE];

// calculate c_sub

__syncthreads();

// write out c_sub to memory
```



NVIDIA CUDA C Programming Guide

Constant Data

- 64KB of “constant” data
  - not written by kernel
- Suitable for read-only, “broadcast” data
- All threads in a warp read the same constant data item at the same time
  - what type of locality is this?
- Uses: Filter coefficients

Summary of data access performance

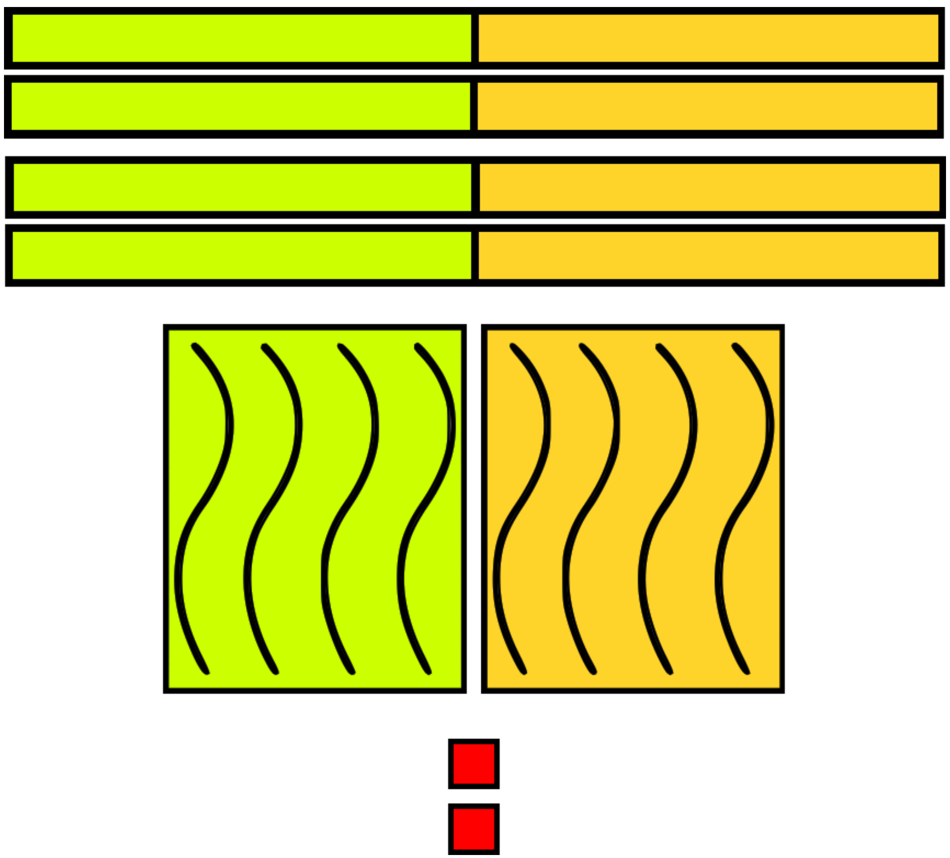
- Layout data structures in memory to maximize bandwidth utilization
- Assign work to threads to maximize bandwidth utilization
- Rethink caching strategies
  - identify readonly data
  - identify blocks that you can load into shared memory
  - identify tables of constants

Distributing Regular Work
Scalar Product

- Problem: Given *n* pairs of vectors, all *w* elements wide, compute the scalar products of all the pairs
  - Multiplications: $n*w$
  - Additions: $n*w$
- How shall we distribute work?

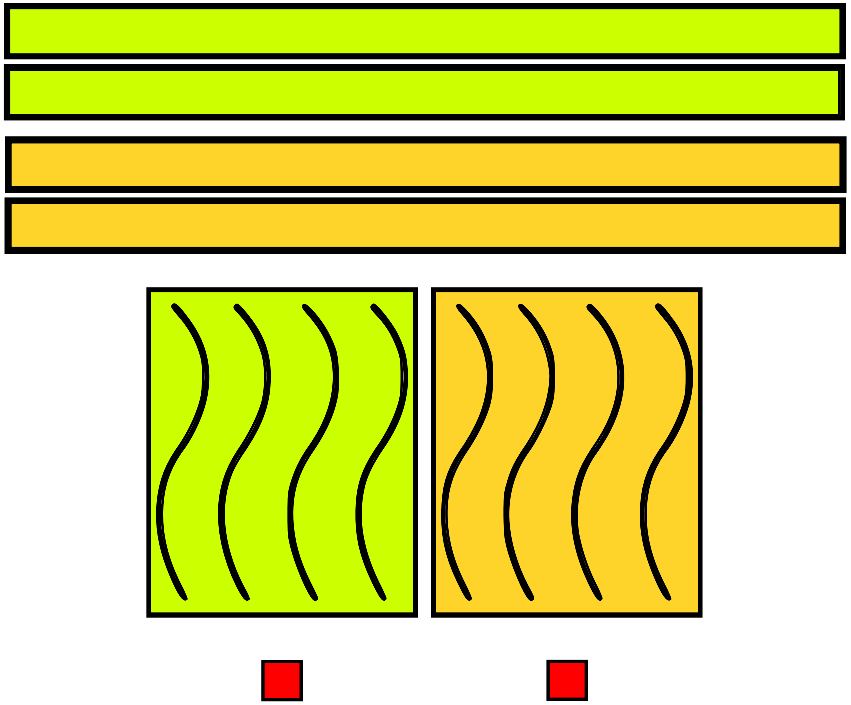
Scheme 1: Split Source Vectors

- Split each vector, and distribute the splits to individual thread blocks

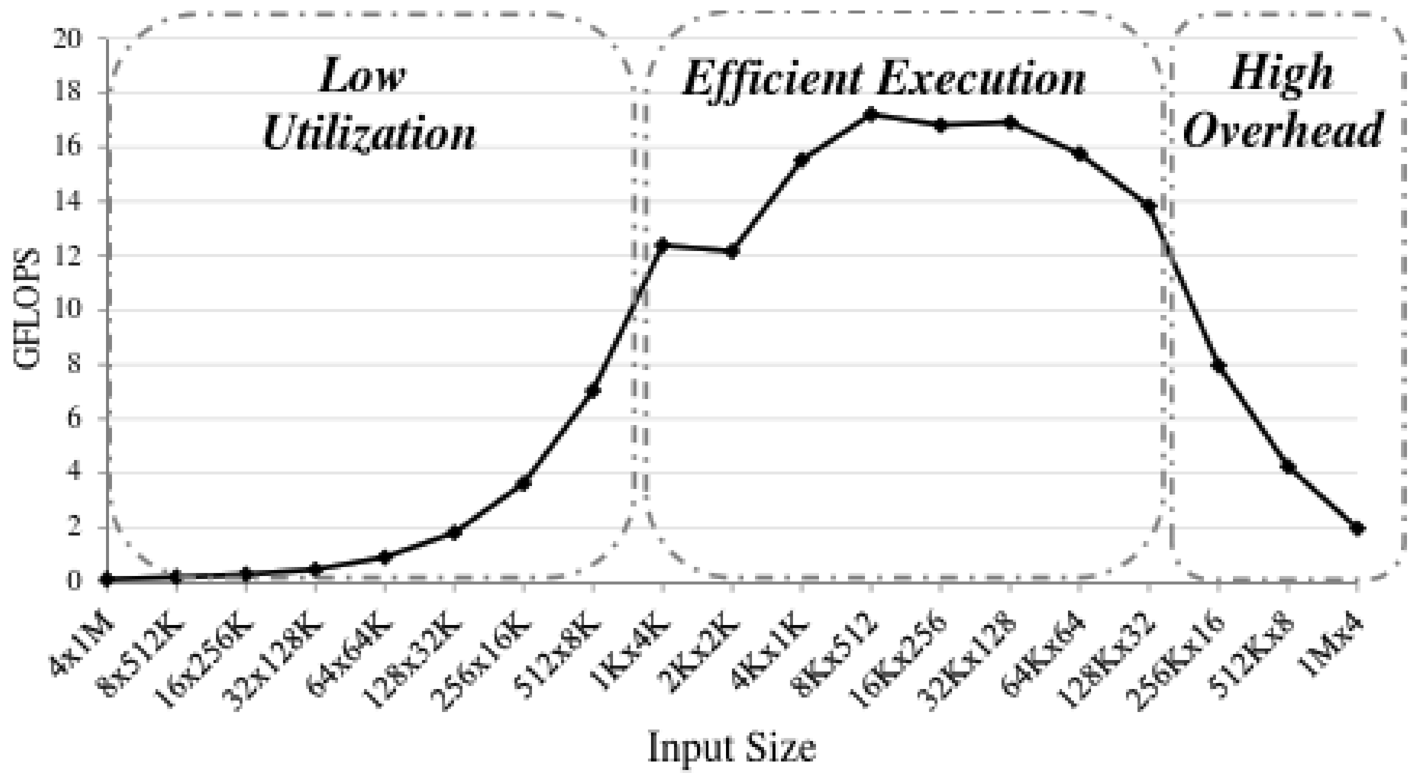


Scheme 2: Split Destination Vector

- Each thread block calculates one scalar product



If only one scheme is used ...



Samadi et al. Adaptive Input-aware Compilation for Graphics Engines, PLDI ’12.

Solution

- Enough work to saturate GPU
- Just not distributed *evenly*
- Make two versions – whole-GPU and per-thread-block
- Choose between two versions at runtime depending on input size
- See MonteCarlo in the CUDA SDK (4.2) for an example

Conclusion

- Focus on full system performance
- Use GPU resources judiciously
  - don't focus on only maximizing occupancy
- Layout data in memory well
  - SoA usually performs better
  - Take advantage of read-only, blocked, and constant characteristics
- Distribute computation well
  - take memory accesses into account
  - be aware of the pitfalls of static scheduling for different input sizes