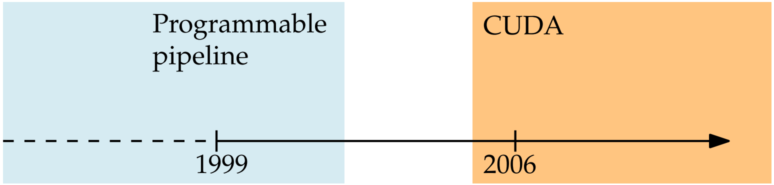
GPU Algorithms III/IV
Computing with CUDA

MADALGO Summer School on Algorithms for Modern Parallel and Distributed Models

Suresh Venkatasubramanian
University of Utah

Previously...



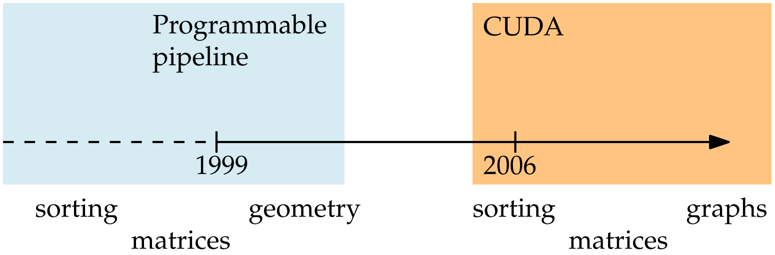
sorting

matrices

geometry

A streaming model

Outline

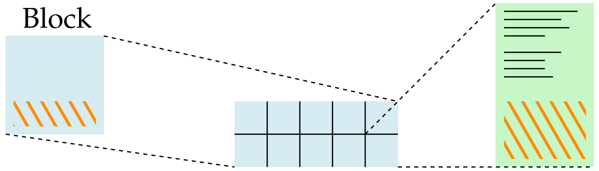


A streaming model

CUDA: Compute Unified Device Architecture

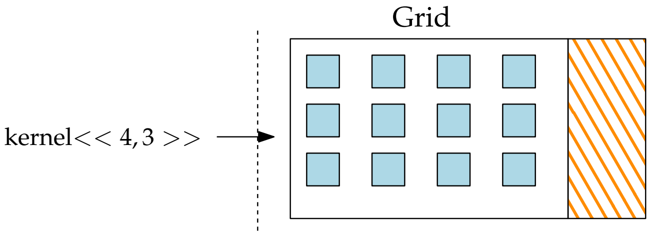
- Lightweight threads that run SIMD (SIMT) in “blocks”
- Blocks run in “SPMD” mode (single program, multiple data)
- Memory at multiple levels (thread, blocks, global)
- Threads are very lightweight, and there are many of them.
- Two views: programmer-centric and hardware-centric

CUDA Model: Blocks



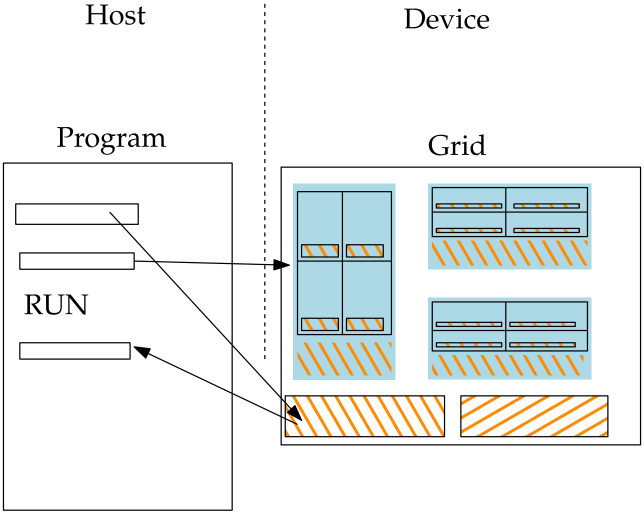
- A block is a collection of threads
- A block can have different "shapes"
- All threads run the same instructions and can synchronize
- Theads have local memory (and so do blocks)
- Block memory is low-latency and shared among threads

CUDA Model: Grids

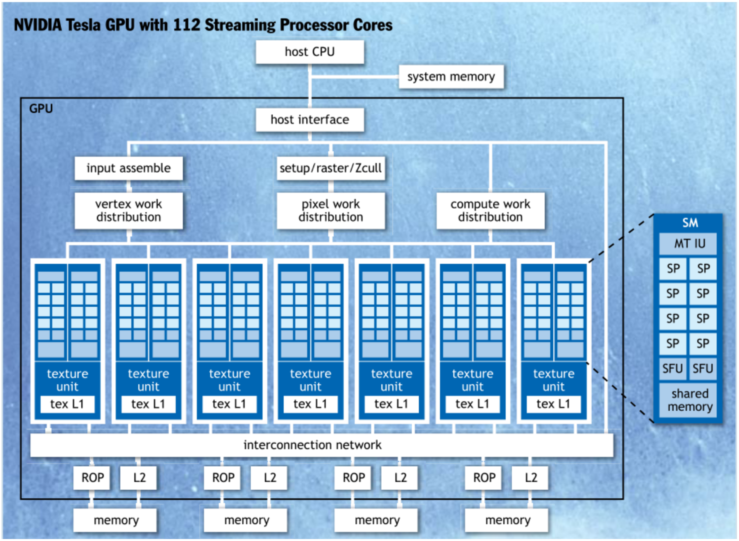


- A grid is a collection of blocks
- A grid can have different shapes
- A grid of blocks is initiated by a request from the host
- A grid has shared memory
- Blocks cannot coordinate with each other and are run independently

CUDA Model: Overview

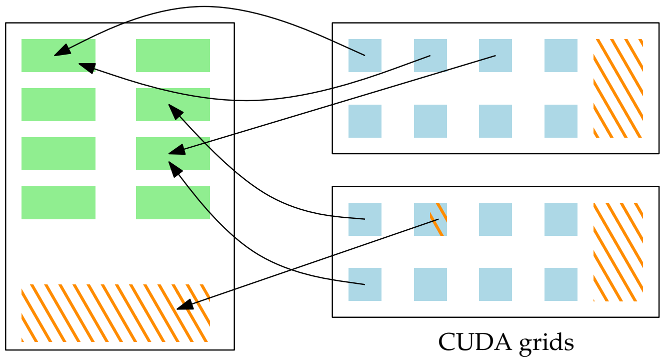


CUDA Execution Model



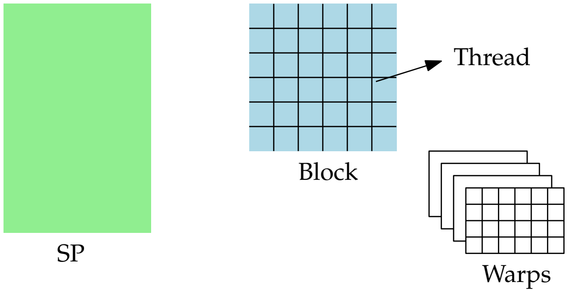
Nickolls, Buck, Garland, Skadron, ACM Queue, Mar 2008[NBGS08]

CUDA Execution Model



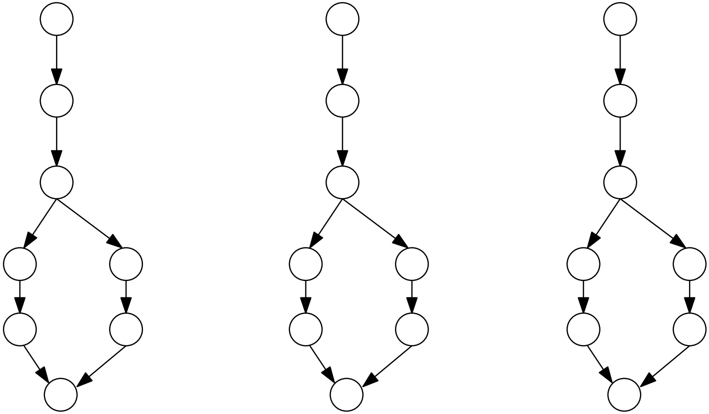
- Each block is assigned to a single SP
- Grid is a software construct
- Block memory managed by SM

CUDA Execution Model

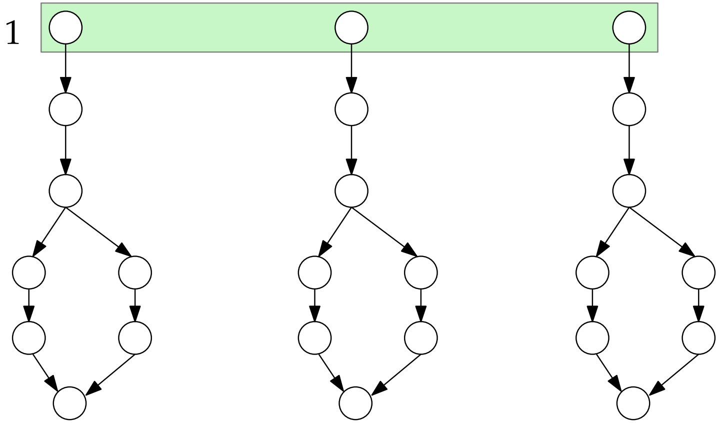


- Each block is divided into groups of 32 threads called ”warps”
- Warp threads are scheduled SIMD on the processor
- Warps are scheduled concurrently

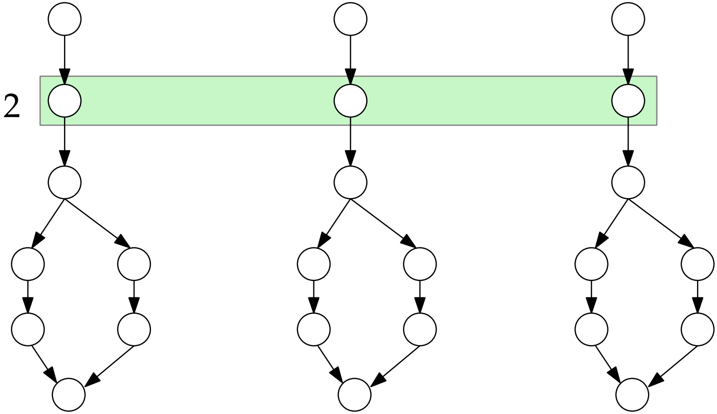
CUDA Design Pitfalls: Branch Divergence



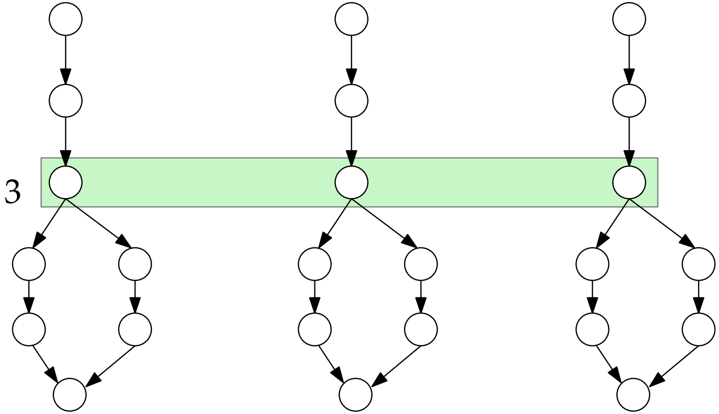
CUDA Design Pitfalls: Branch Divergence



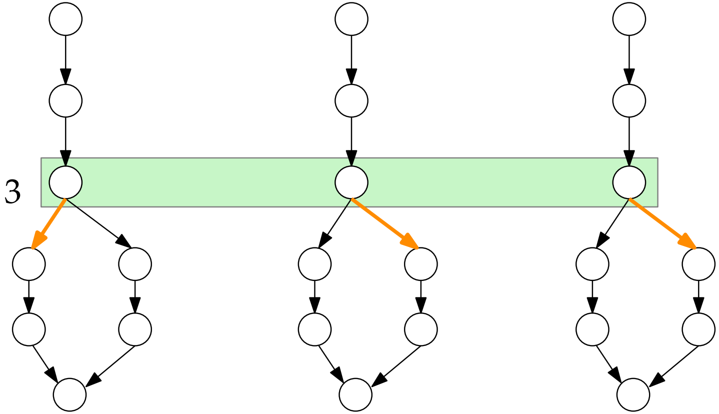
CUDA Design Pitfalls: Branch Divergence



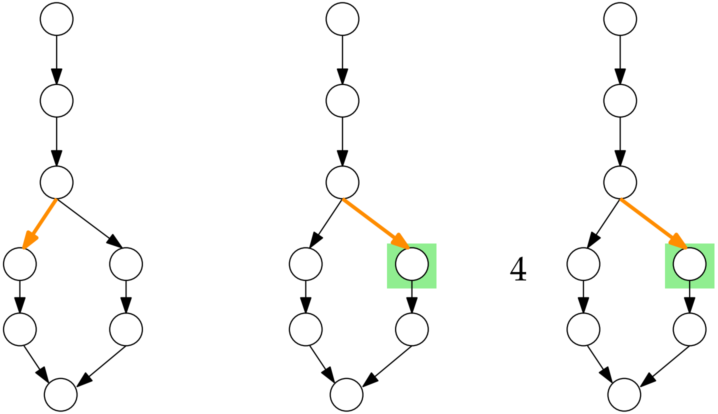
CUDA Design Pitfalls: Branch Divergence



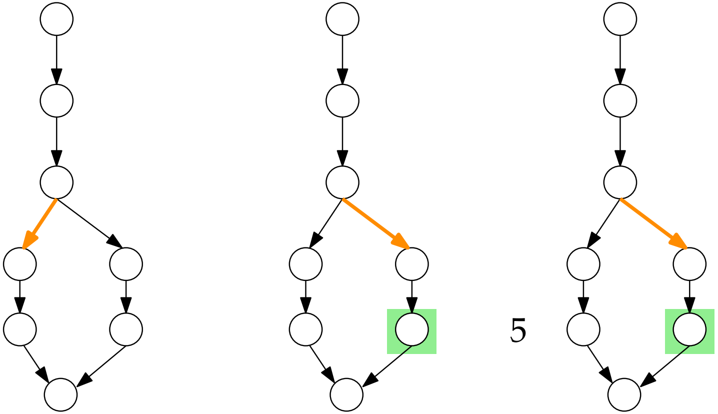
CUDA Design Pitfalls: Branch Divergence



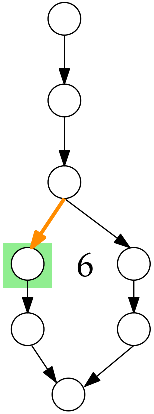
CUDA Design Pitfalls: Branch Divergence

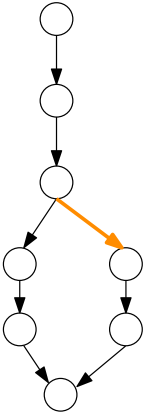
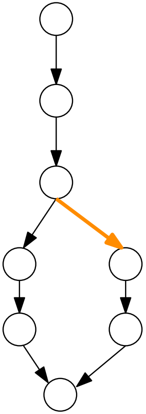


CUDA Design Pitfalls: Branch Divergence

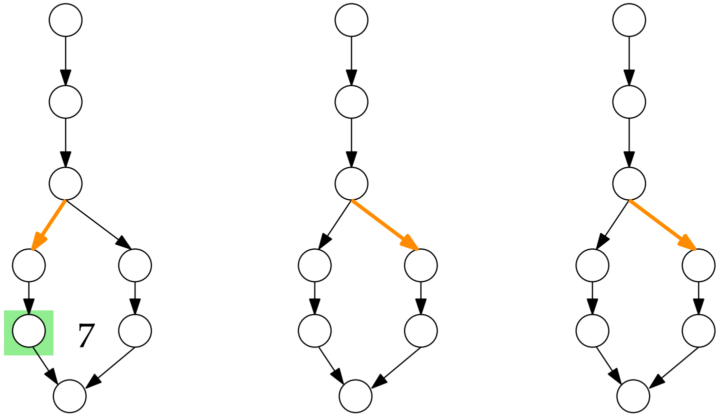


CUDA Design Pitfalls: Branch Divergence

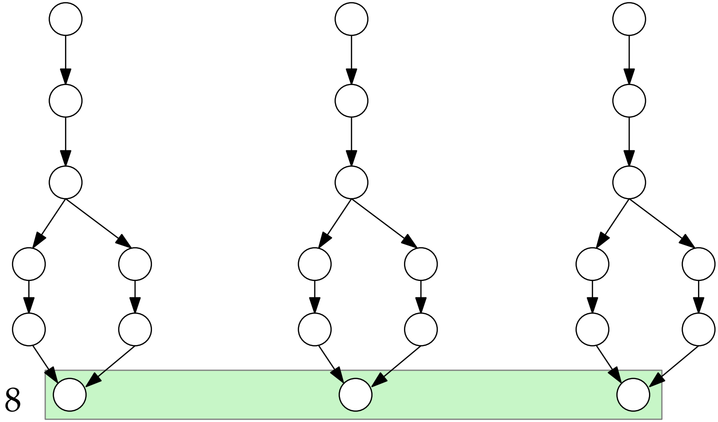


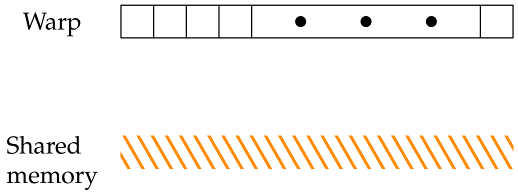
CUDA Design Pitfalls: Branch Divergence



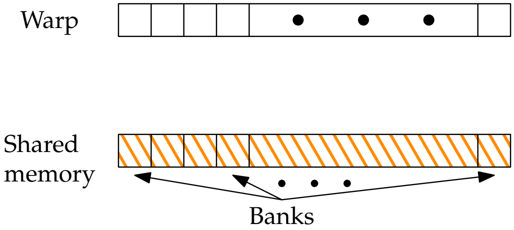
CUDA Design Pitfalls: Branch Divergence



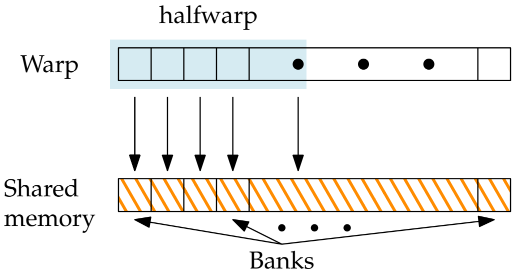
CUDA Design Pitfalls: Memory Bank Conflicts



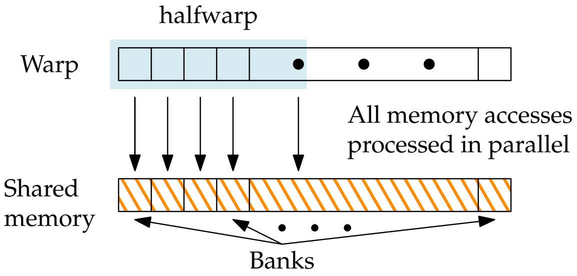
CUDA Design Pitfalls: Memory Bank Conflicts



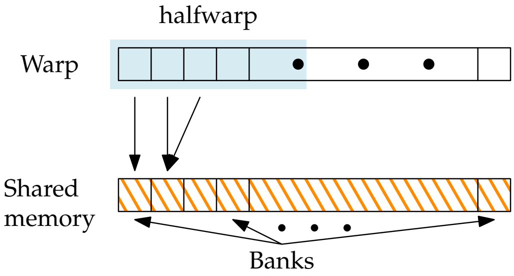
CUDA Design Pitfalls: Memory Bank Conflicts



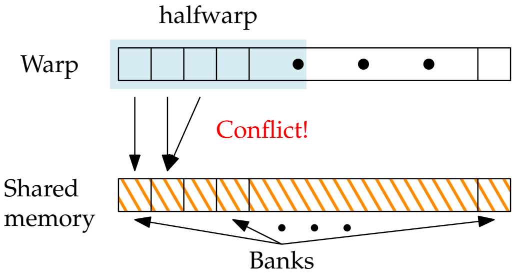
CUDA Design Pitfalls: Memory Bank Conflicts



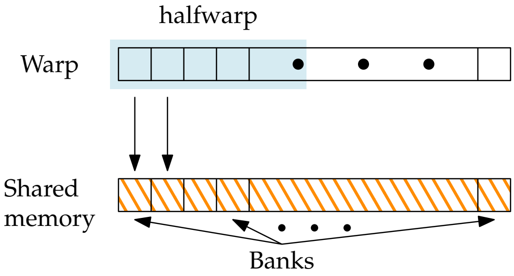
CUDA Design Pitfalls: Memory Bank Conflicts



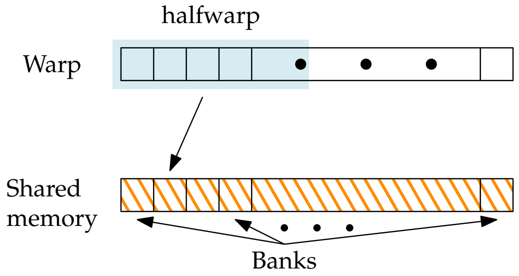
CUDA Design Pitfalls: Memory Bank Conflicts



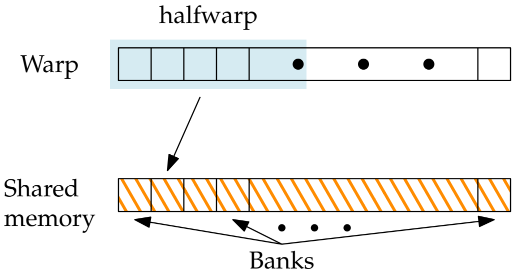
CUDA Design Pitfalls: Memory Bank Conflicts



CUDA Design Pitfalls: Memory Bank Conflicts

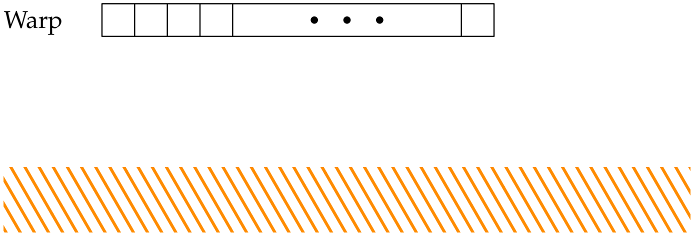


CUDA Design Pitfalls: Memory Bank Conflicts

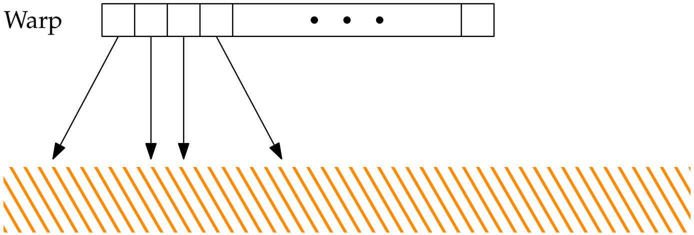


Memory bank conflicts can result in serialized access

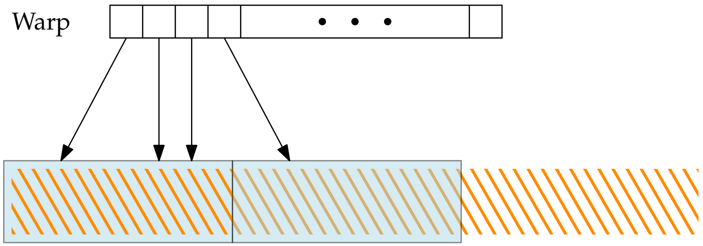
CUDA Design Pitfalls: Global Memory Coalescing



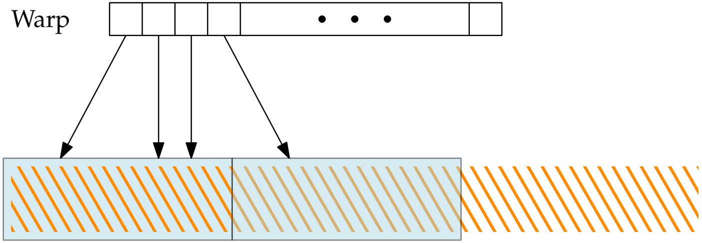
CUDA Design Pitfalls: Global Memory Coalescing



CUDA Design Pitfalls: Global Memory Coalescing

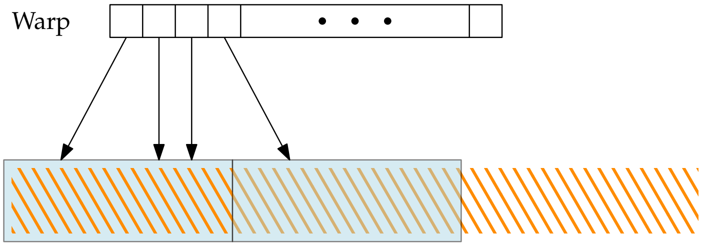


CUDA Design Pitfalls: Global Memory Coalescing



Both cache blocks are returned

CUDA Design Pitfalls: Global Memory Coalescing

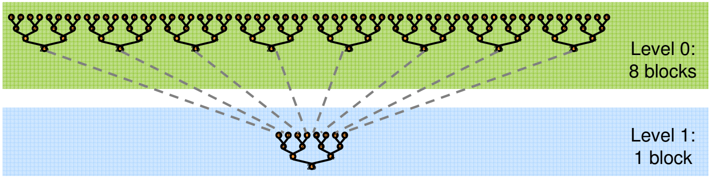


Both cache blocks are returned

- Threads should ”coalesce” access to single memory line
- This is akin to block transfer in external memory

Solution: Kernel Decomposition

- Avoid global sync by decomposing computation into multiple kernel invocations



- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation

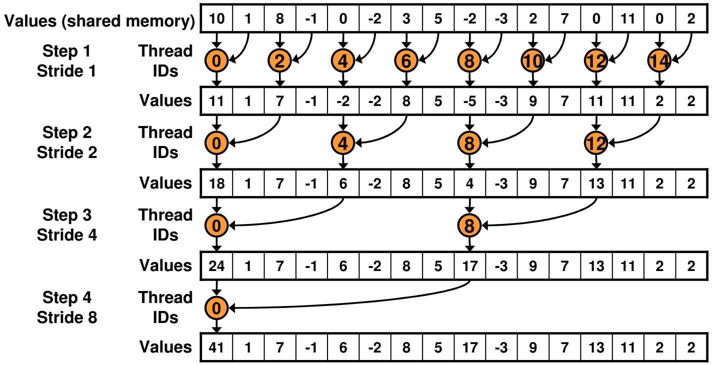
Mark Harris. Optimizing Parallel Reduction in CUDA.[HBM$^+$07, SHZO07]

Reduction #1: Interleaved Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {
   extern __shared__ int sdata[];

   // each thread loads one element from global to shared mem
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
   sdata[tid] = g_idata[i];
   __syncthreads();

   // do reduction in shared mem
   for(unsigned int s=1; s < blockDim.x; s *= 2) {
      if (tid % (2*s) == 0) {
         sdata[tid] += sdata[tid + s];
      }
      __syncthreads();
   }

   // write result for this block to global mem
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Mark Harris. Optimizing Parallel Reduction in CUDA.[HBM^{+}07, SHZO07]

Parallel Reduction: Interleaved Addressing



Mark Harris. Optimizing Parallel Reduction in CUDA.[HBM$^+$07, SHZO07]

Reduction #1: Interleaved Addressing

```
__global__ void reduce1(int *g_idata, int *g_odata) {
   extern __shared__ int sdata[];

   // each thread loads one element from global to shared mem
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
   sdata[tid] = g_idata[i];
   __syncthreads();

   // do reduction in shared mem
   for (unsigned int s=1; s < blockDim.x; s *= 2) {
      if (tid % (2*s) == 0) {
         sdata[tid] += sdata[tid + s];
      }
      __syncthreads();
   }

   // write result for this block to global mem
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent warps are very inefficient, and % operator is very slow

Mark Harris. Optimizing Parallel Reduction in CUDA.[HBM$^+$07, SHZO07]

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s

Note: Block Size = 128 threads for all tests

Reduction #2: Interleaved Addressing

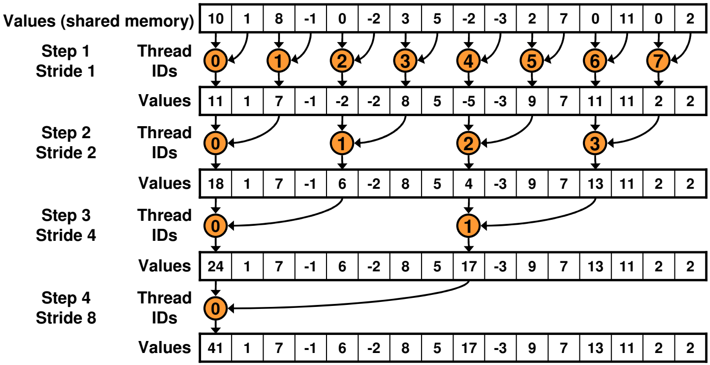
Just replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

Parallel Reduction: Interleaved Addressing



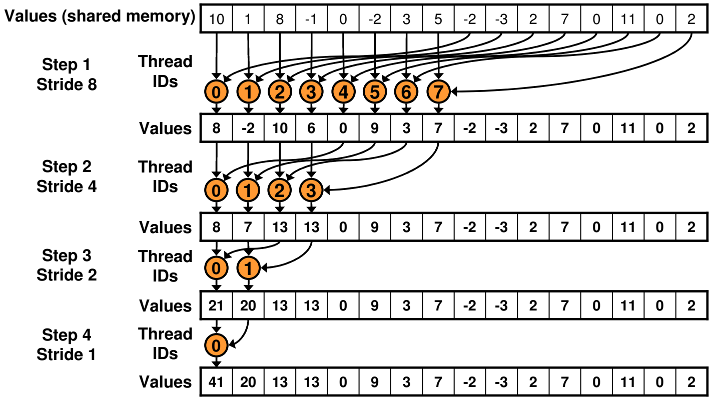
New Problem: Shared Memory Bank Conflicts

Mark Harris. Optimizing Parallel Reduction in CUDA.[HBM+07, SHZO07]

Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x

Parallel Reduction: Sequential Addressing



Sequential addressing is conflict free

Mark Harris. Optimizing Parallel Reduction in CUDA.[HBM$^+$07, SHZO07]

Reduction #3: Sequential Addressing

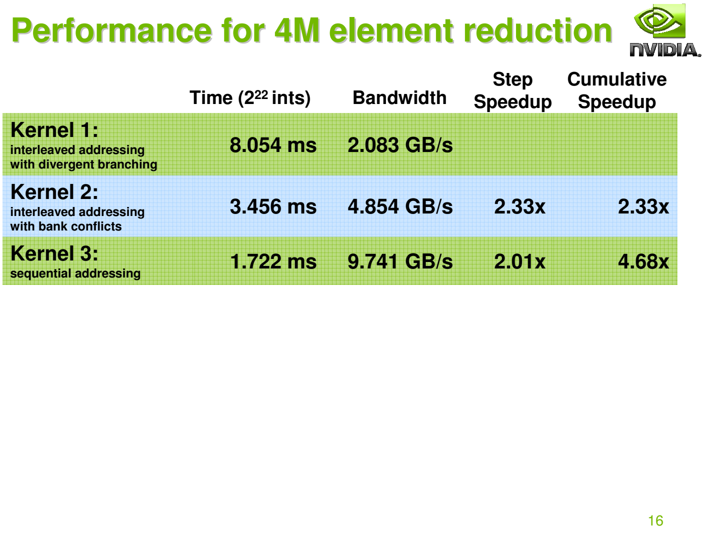
Just replace strided indexing in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

With reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Mark Harris. Optimizing Parallel Reduction in CUDA.[HBM+07, SHZO07]



Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

16

Mark Harris. Optimizing Parallel Reduction in CUDA.[HBM+07, SHZO07]

Applications

Sample Sort[LOS10]

1 23 7 19 25 42 4

Sample Sort[LOS10]

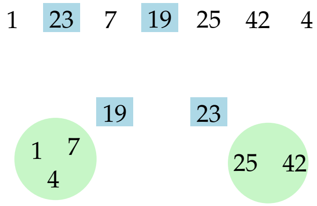
1 23 7 19 25 42 4

Sample Sort[LOS10]

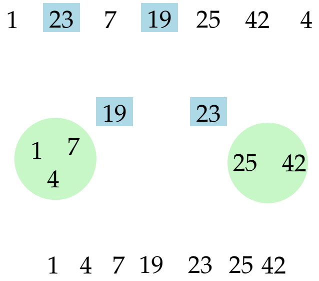
1 23 7 19 25 42 4

19 23

Sample Sort[LOS10]



Sample Sort[LOS10]



Sample Sort

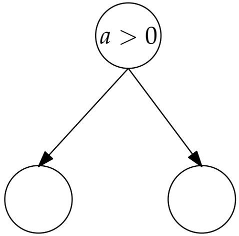
```
Given $X = \{x_1, x_2, \ldots x_n\}, k$
  **if** $n \le M$ **then**
    SimpleSort$(X)$
  **end if**
  Pick random sample $R = x_{i_1}, x_{i_2}, \ldots x_{i_k}$
  Sort$(R) = \{r_0 = -\infty, r_1, r_2, \ldots, r_k, +\infty = r_{k+1}\}$
  Place $x_i$ in bucket $b_j$ if $r_j \le x_i < r_{j+1}$
  Concatenate SampleSort$(b_0, k)$, SampleSort$(b_1, k)$, $\ldots$
```

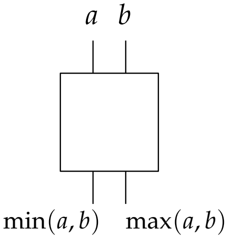
- Parallelize the individual sorts
- Use parallel reductions to partition elements

GPU Algorithm: A single phase

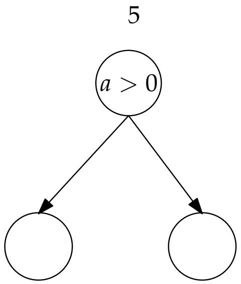
Phase 1	Compute the sample and sort it
Phase 2	Within each block, figure out the bucket indices for each element. Construct k-element histogram. Copy to global memory
Phase 3	Do prefix sum (parallel reduction!) to find global offsets
Phase 4	Distribute items using global offset

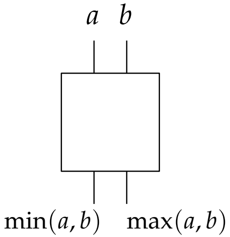
Data-parallel Binary Search



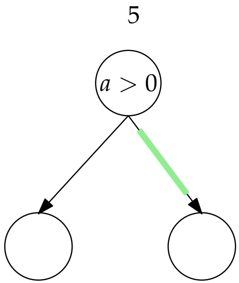


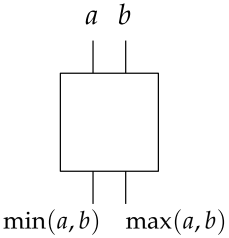
Data-parallel Binary Search



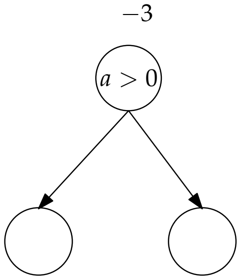


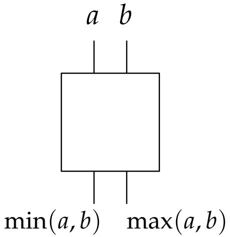
Data-parallel Binary Search



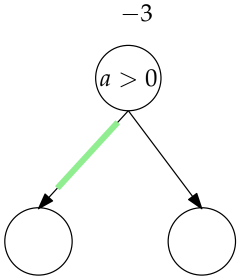


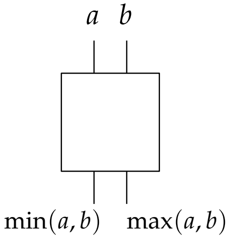
Data-parallel Binary Search



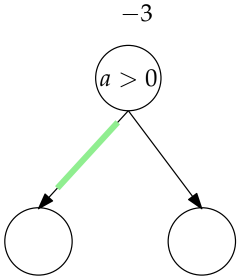


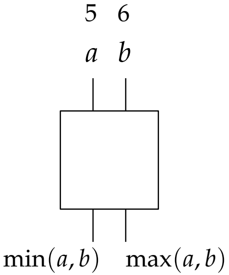
Data-parallel Binary Search



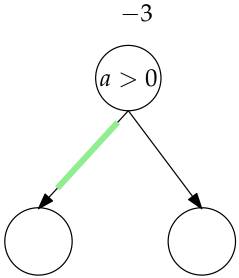


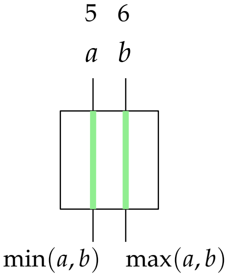
Data-parallel Binary Search



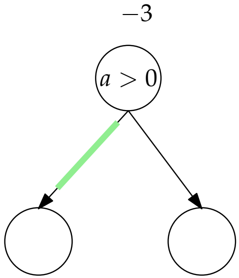


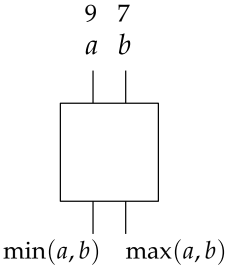
Data-parallel Binary Search



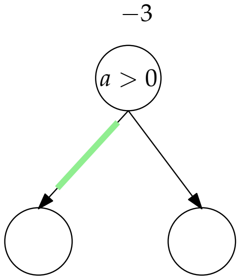


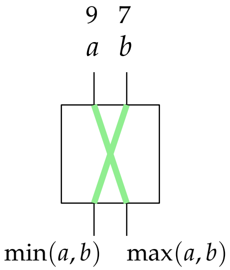
Data-parallel Binary Search



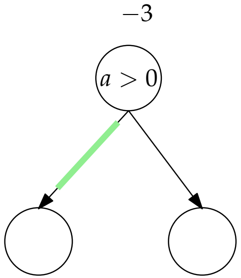


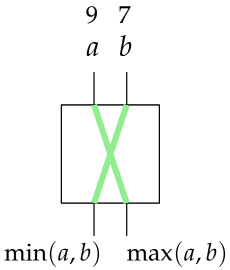
Data-parallel Binary Search





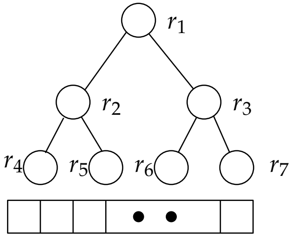
Data-parallel Binary Search



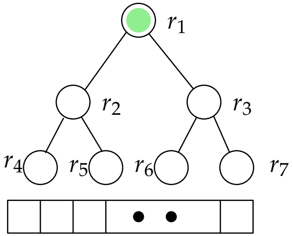


- In first comparator, a path is taken or not.
- In second, both paths are used, but by different data

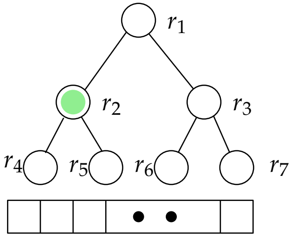
Predicated Instructions



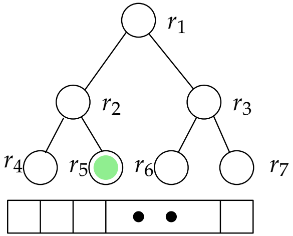
Predicated Instructions



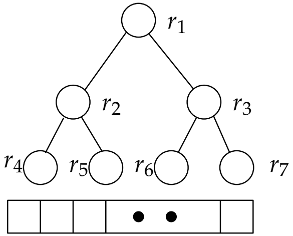
Predicated Instructions



Predicated Instructions



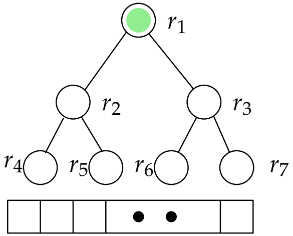
Predicated Instructions



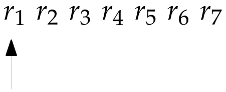
```
$j = 1$
for $i = 1$ to $\log 4$ do
    $j \leftarrow 2j + (q > r_i)$
end for
$j \leftarrow j - 4 + 1$
```

$r_1\ r_2\ r_3\ r_4\ r_5\ r_6\ r_7$

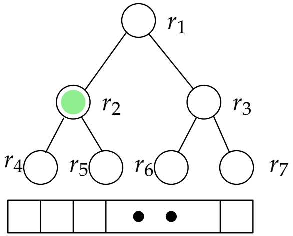
Predicated Instructions



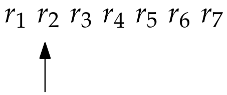
```
$j = 1$
**for** $i = 1$ to $\log 4$ **do**
    $j \leftarrow 2j + (q > r_i)$
**end for**
$j \leftarrow j - 4 + 1$
```



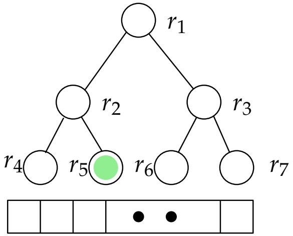
Predicated Instructions



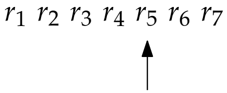
```
$j = 1$
**for** $i = 1$ to $\log 4$ **do**
    $j \leftarrow 2j + (q > r_i)$
**end for**
$j \leftarrow j - 4 + 1$
```



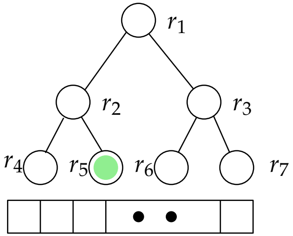
Predicated Instructions



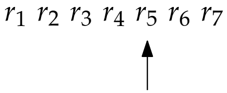
```
$j = 1$
for $i = 1$ to $\log 4$ do
    $j \leftarrow 2j + (q > r_i)$
end for
$j \leftarrow j - 4 + 1$
```



Predicated Instructions

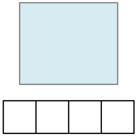


```
$j = 1$
for $i = 1$ to $\log 4$ do
    $j \leftarrow 2j + (q > r_i)$
end for
$j \leftarrow j - 4 + 1$
```

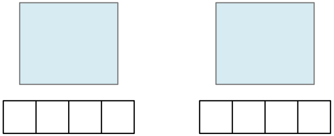


Conditionals are replaced by predicated statements

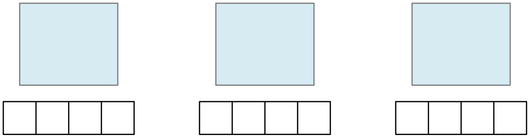
Parallel Prefix Sum for Offsets



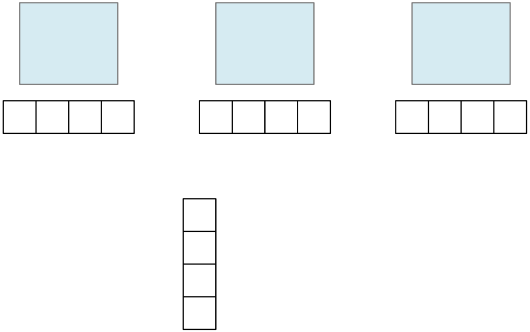
Parallel Prefix Sum for Offsets



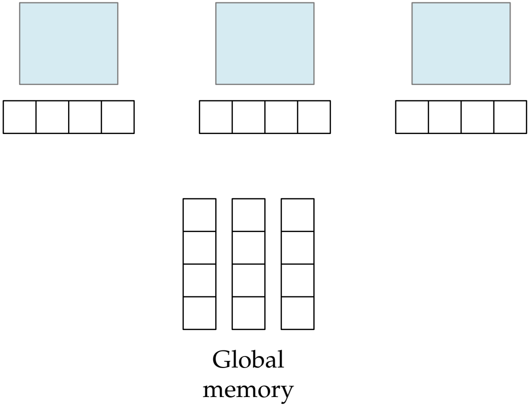
Parallel Prefix Sum for Offsets



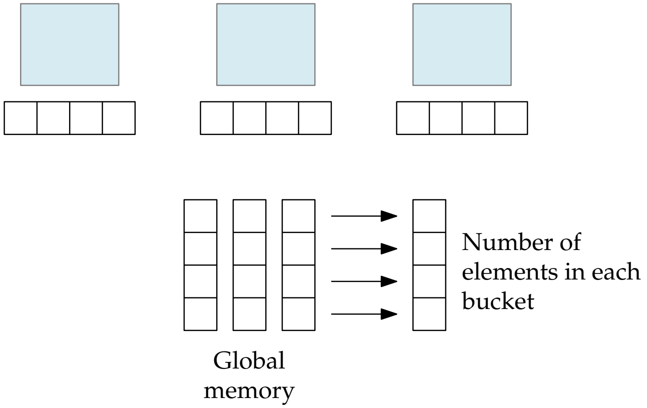
Parallel Prefix Sum for Offsets



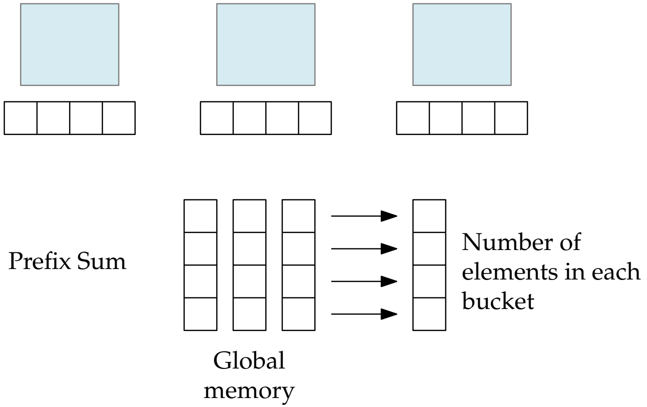
Parallel Prefix Sum for Offsets



Parallel Prefix Sum for Offsets

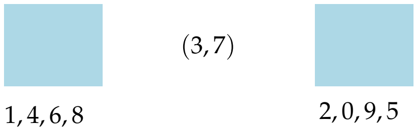


Parallel Prefix Sum for Offsets

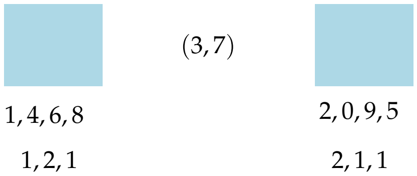


Distribution

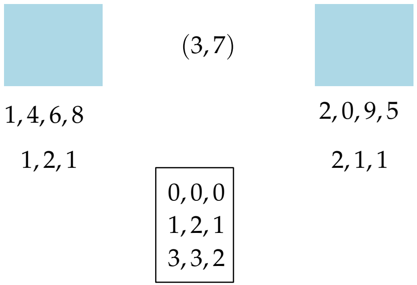
Distribution



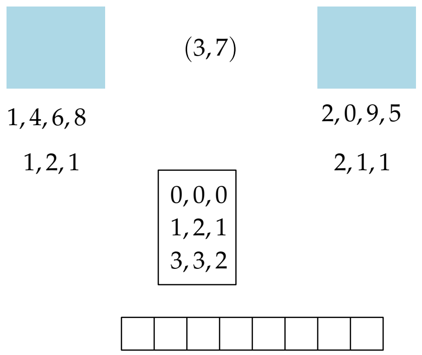
Distribution



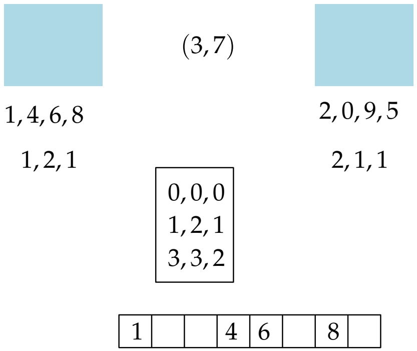
Distribution



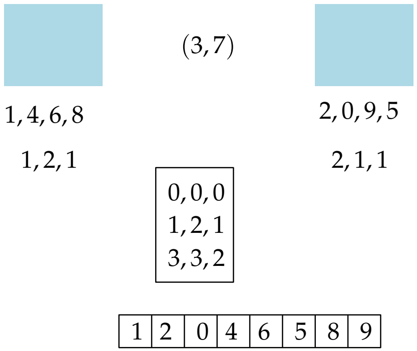
Distribution



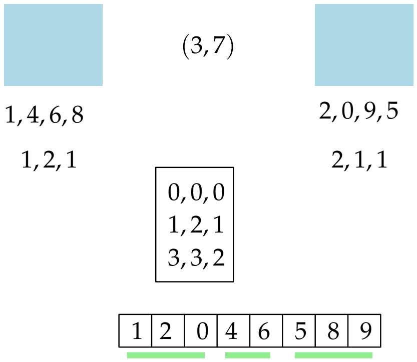
Distribution



Distribution

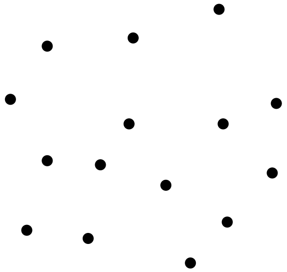


Distribution

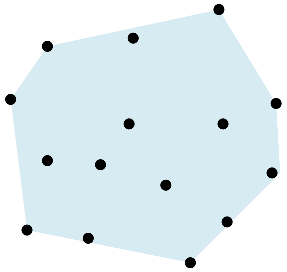


Repeat in each block

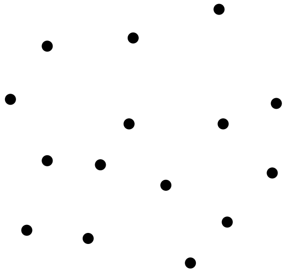
Convex Hulls



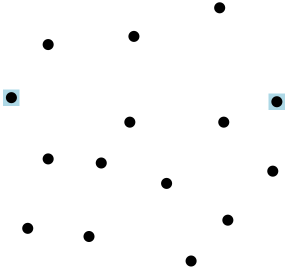
Convex Hulls



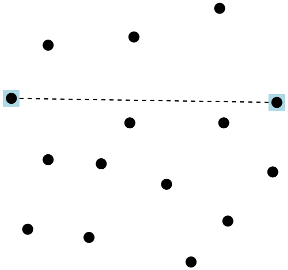
Convex Hulls



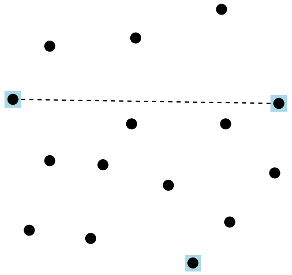
Convex Hulls



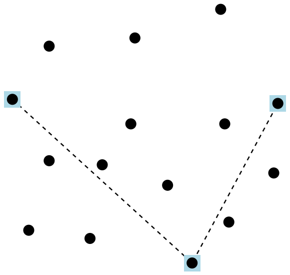
Convex Hulls



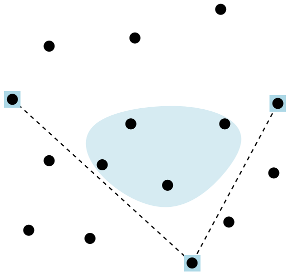
Convex Hulls



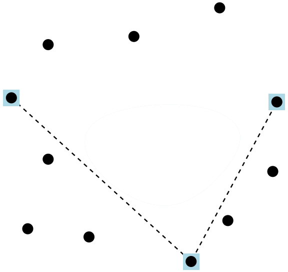
Convex Hulls



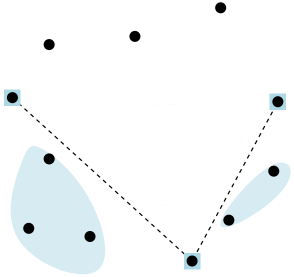
Convex Hulls



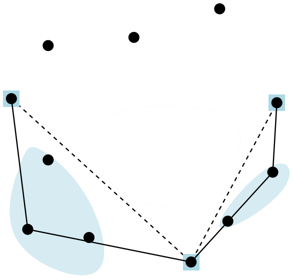
Convex Hulls



Convex Hulls



Convex Hulls



GPU QuickHull[SKGN09]

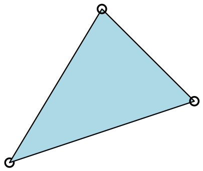
Phase 1	Compute the sample and sort it
Phase 2	Within each block, figure out the bucket indices for each element. Construct *k*-element histogram. Copy to global memory
Phase 3	Do prefix sum (parallel reduction!) to find global offsets
Phase 4	Distribute items using global offset

GPU QuickHull[SKGN09]

Phase 1	Compute the pivot point
Phase 2	Within each block, eliminate elements above the pivot segment. Store the rest
Phase 3	Do prefix sum (parallel reduction!) to find global offsets
Phase 4	Distribute items using global offset

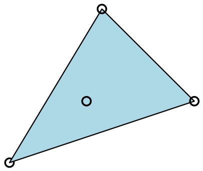
3D Quick Hull Algorithm

- Instead of line segments, the separating objects are planes.
- As before, points are distributed to the planes that they are "outside" of.
- The operation of extending the hull can create "concave" edges that need to be repaired.
- This is a hybrid CPU-GPU algorithm: distribution happens on the GPU, and the rest happens on the CPU.



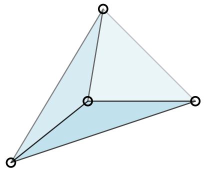
3D Quick Hull Algorithm

- Instead of line segments, the separating objects are planes.
- As before, points are distributed to the planes that they are “outside” of.
- The operation of extending the hull can create “concave” edges that need to be repaired.
- This is a hybrid CPU-GPU algorithm: distribution happens on the GPU, and the rest happens on the CPU.



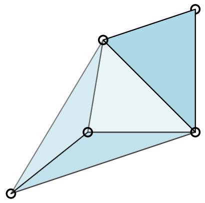
3D Quick Hull Algorithm

- Instead of line segments, the separating objects are planes.
- As before, points are distributed to the planes that they are "outside" of.
- The operation of extending the hull can create "concave" edges that need to be repaired.
- This is a hybrid CPU-GPU algorithm: distribution happens on the GPU, and the rest happens on the CPU.



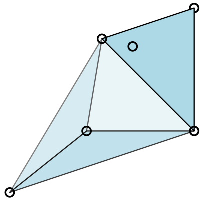
3D Quick Hull Algorithm

- Instead of line segments, the separating objects are planes.
- As before, points are distributed to the planes that they are "outside" of.
- The operation of extending the hull can create "concave" edges that need to be repaired.
- This is a hybrid CPU-GPU algorithm: distribution happens on the GPU, and the rest happens on the CPU.



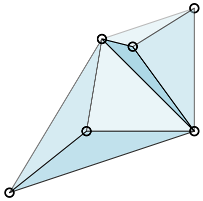
3D Quick Hull Algorithm

- Instead of line segments, the separating objects are planes.
- As before, points are distributed to the planes that they are "outside" of.
- The operation of extending the hull can create "concave" edges that need to be repaired.
- This is a hybrid CPU-GPU algorithm: distribution happens on the GPU, and the rest happens on the CPU.



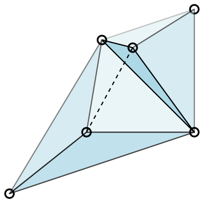
3D Quick Hull Algorithm

- Instead of line segments, the separating objects are planes.
- As before, points are distributed to the planes that they are "outside" of.
- The operation of extending the hull can create "concave" edges that need to be repaired.
- This is a hybrid CPU-GPU algorithm: distribution happens on the GPU, and the rest happens on the CPU.



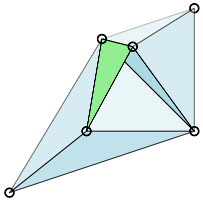
3D Quick Hull Algorithm

- Instead of line segments, the separating objects are planes.
- As before, points are distributed to the planes that they are "outside" of.
- The operation of extending the hull can create "concave" edges that need to be repaired.
- This is a hybrid CPU-GPU algorithm: distribution happens on the GPU, and the rest happens on the CPU.



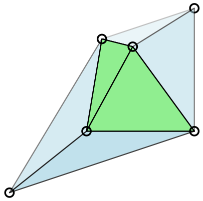
3D Quick Hull Algorithm

- Instead of line segments, the separating objects are planes.
- As before, points are distributed to the planes that they are "outside" of.
- The operation of extending the hull can create "concave" edges that need to be repaired.
- This is a hybrid CPU-GPU algorithm: distribution happens on the GPU, and the rest happens on the CPU.

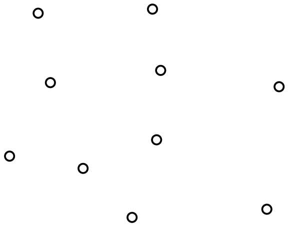


3D Quick Hull Algorithm

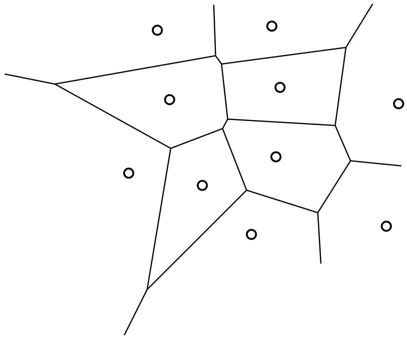
- Instead of line segments, the separating objects are planes.
- As before, points are distributed to the planes that they are "outside" of.
- The operation of extending the hull can create "concave" edges that need to be repaired.
- This is a hybrid CPU-GPU algorithm: distribution happens on the GPU, and the rest happens on the CPU.



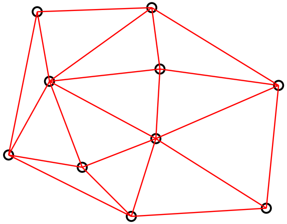
Lifting map



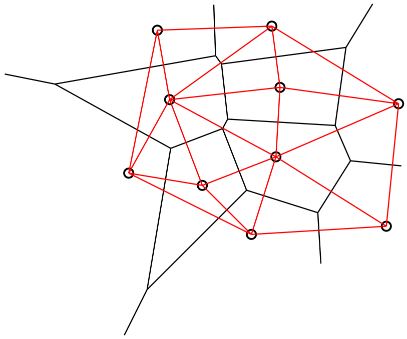
Lifting map



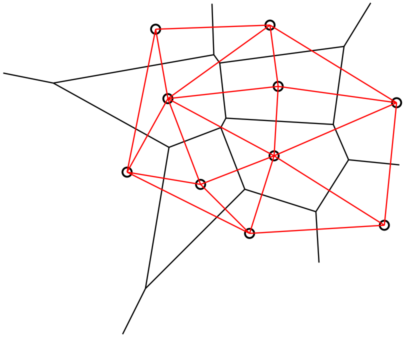
Lifting map

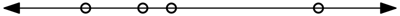


Lifting map

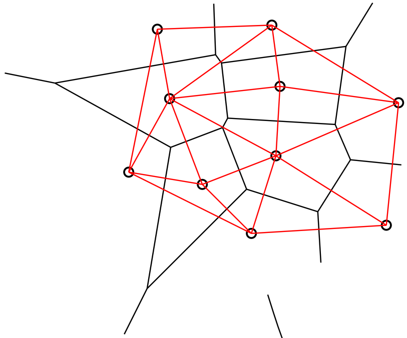


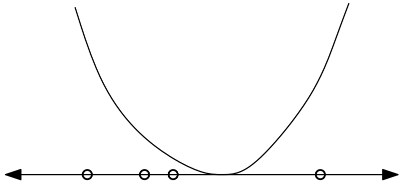
Lifting map



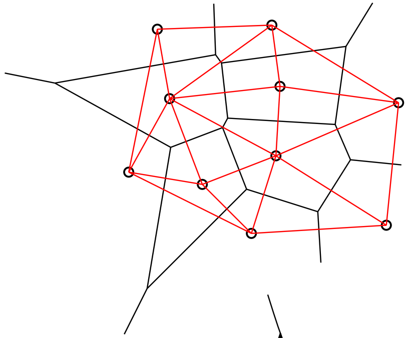


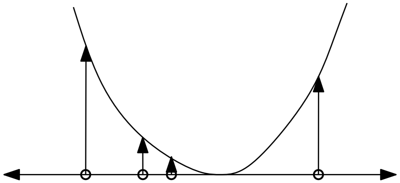
Lifting map



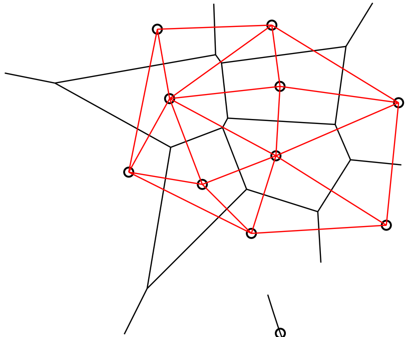


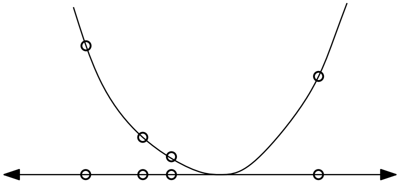
Lifting map



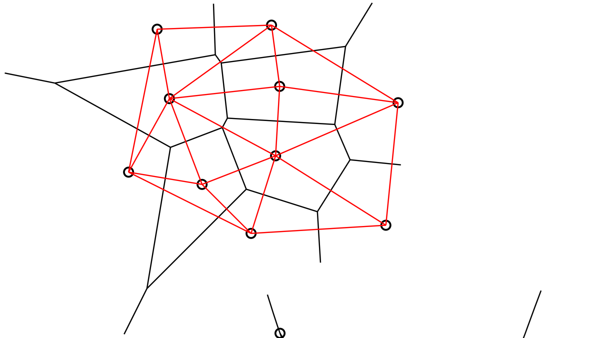


Lifting map

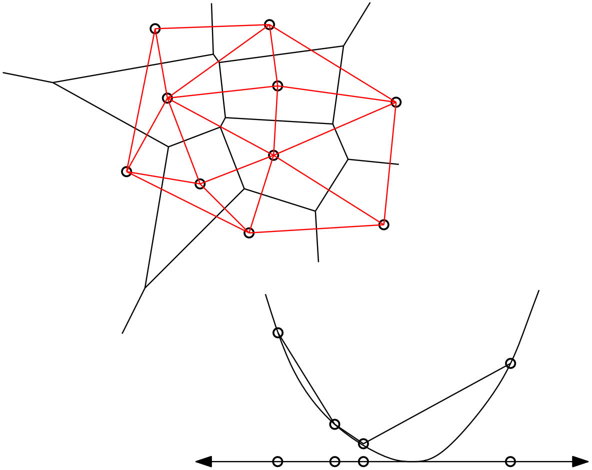




Lifting map

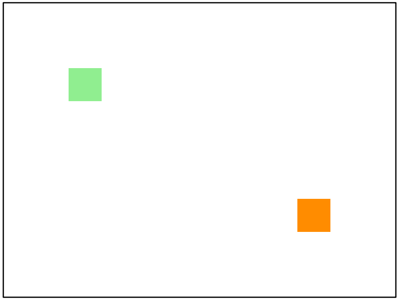


Lifting map

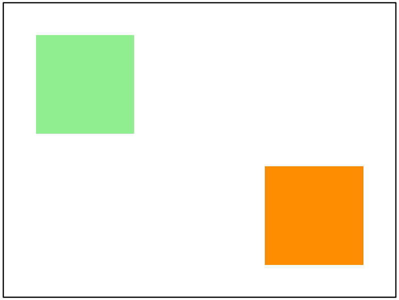


3D convex hull ⇔ 2D Delaunay ⇔ 2D Voronoi

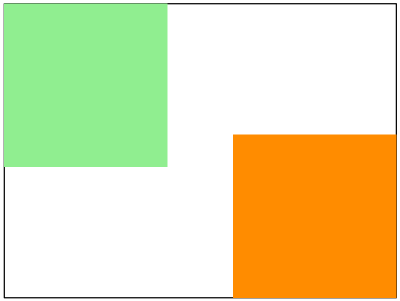
Jump Flooding algorithm[RT06]



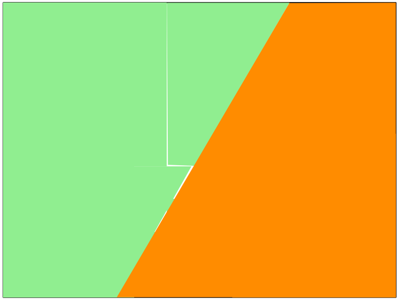
Jump Flooding algorithm[RT06]



Jump Flooding algorithm[RT06]



Jump Flooding algorithm[RT06]

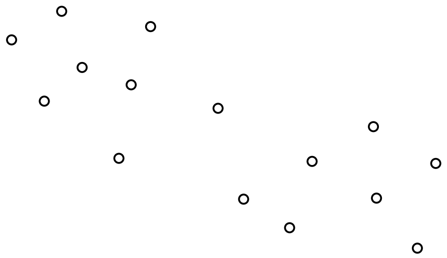


Challenge

Challenge

Can we get better algorithms for computing 2D Voronoi diagrams and lower envelopes ?

k-means clustering[ZG09]

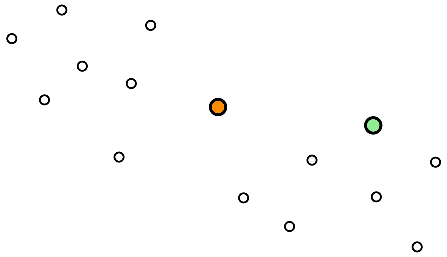


Find k centers $\mathcal{C} = c_1, \ldots c_k$ such that

$$\sum_{p \in P} \min_{c \in C} \|p - c\|^2$$

is minimized

k-means clustering[ZG09]

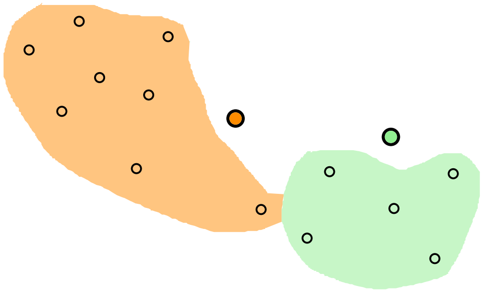


Find k centers $\mathcal{C} = c_1, \ldots c_k$ such that

$$\sum_{p \in P} \min_{c \in C} \|p - c\|^2$$

is minimized

k-means clustering[ZG09]

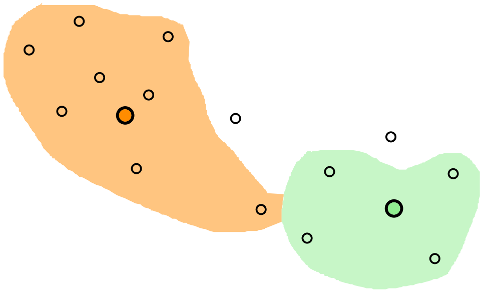


Find k centers $\mathcal{C} = c_1, \ldots c_k$ such that

$$\sum_{p \in P} \min_{c \in C} \|p - c\|^2$$

is minimized

k-means clustering[ZG09]

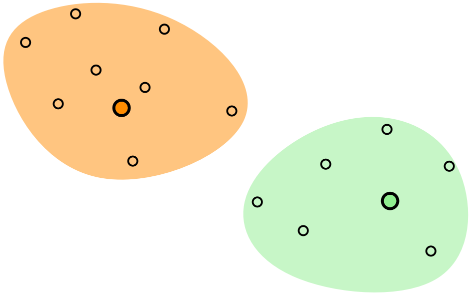


Find k centers $\mathcal{C} = c_1, \dots c_k$ such that

$$\sum_{p \in P} \min_{c \in C} \|p - c\|^2$$

is minimized

k-means clustering[ZG09]



Find k centers $\mathcal{C} = c_1, \dots c_k$ such that

$$\sum_{p \in P} \min_{c \in C} \|p - c\|^2$$

is minimized

Where are the expensive computations ?

- Each point finds its nearest neighbor ($O(nk)$ or $O(n \log k)$ if clever)
- We compute the centroids of points in clusters
- Points are fixed in all iterations, but centroids change.

Standard Implementation: CPU-GPU hybrid

- For each point, need to compute $\min_{\mathcal{C}} \|p - c\|$
- This is a trivial parallelization
    - One thread for each point (or block appropriately)
    - In each iteration, compute distances from a single center.
    - $k \ll n$, so this is not expensive
- For each labelling, find new center.
    - Could do this entirely in GPU: centers computed via reduce operation.
    - Most algorithms don't do it: copy to CPU.

Why irregular access is a problem

- GPUs are designed for high arithmetic intensity and SIMT behavior
- Irregular data locality and access (such as with graphs) reduces the benefit of these methods
- To handle sparse data, you need to store the data compactly, and process it efficiently based on the format.

Sparse Matrix-Vector multiplication

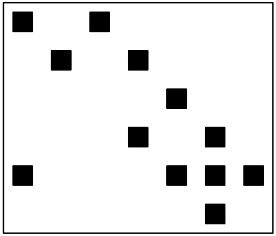
$$y = Ax$$

- Easy if A is dense using kernel at each vector: $O(n^2)$
- If number of nonzeros in A is small, would prefer $O(\text{nnz}(A))$ (or linear in input)

Representations

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

Diagonal form



$$\texttt{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}$$

$$\texttt{offsets} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

Representations

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$$

$$\texttt{indices} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$

$$\texttt{data} = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

CSR representation

GPU SpMV I

$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$

$\texttt{indices} =$
$\begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$

$\texttt{data} =$
$\begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$

CSR representation

```
start ← $ptr[ID]$
end ← $ptr[ID + 1]$
**for** $i$ = start to end **do**
    d = d + data[i] + x[indices[start]]
**end for**
```

GPU SpMV I: Access Patterns

Threads in this kernel access elements haphazardly:

$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$

$$\begin{aligned} \texttt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \texttt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

Round 1:

Round 2:

Round 3:

GPU SpMV I: Access Patterns

Threads in this kernel access elements haphazardly:

$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$

$\texttt{indices} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$

$\texttt{data} = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$

Round 1: ×

Round 2:

Round 3:

GPU SpMV I: Access Patterns

Threads in this kernel access elements haphazardly:

$$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$$

$$\texttt{indices} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$

$$\texttt{data} = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

Round 1: × ×

Round 2:

Round 3:

GPU SpMV I: Access Patterns

Threads in this kernel access elements haphazardly:

$$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$$

$$\texttt{indices} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$

$$\texttt{data} = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

Round 1: × × ×

Round 2:

Round 3:

GPU SpMV I: Access Patterns

Threads in this kernel access elements haphazardly:

$$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$$

$$\texttt{indices} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$

$$\texttt{data} = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

Round 1:	×		×		×			×	
Round 2:									
Round 3:									

GPU SpMV I: Access Patterns

Threads in this kernel access elements haphazardly:

$$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$$

$$\texttt{indices} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$

$$\texttt{data} = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

Round 1:	×		×		×			×	
Round 2:		×		×		×			×
Round 3:									

GPU SpMV I: Access Patterns

Threads in this kernel access elements haphazardly:

$$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$$

$$\texttt{indices} = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$

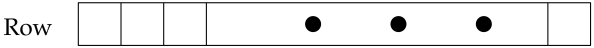
$$\texttt{data} = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

Round 1:	×		×		×			×	
Round 2:		×		×		×			×
Round 3:							×		

GPU SpMV II

New idea: instead of assigning one thread per row, assign one warp per row.

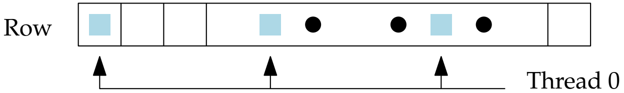
- Each thread in a warp sums up a piece of the dot product
- At end, a parallel reduction combines the pieces of the sum
- Unrolling helps speed the reduction.



GPU SpMV II

New idea: instead of assigning one thread per row, assign one warp per row.

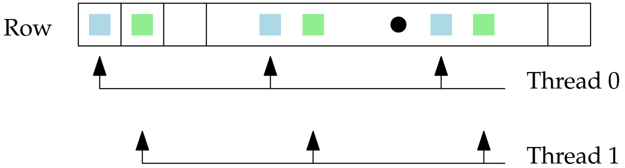
- Each thread in a warp sums up a piece of the dot product
- At end, a parallel reduction combines the pieces of the sum
- Unrolling helps speed the reduction.



GPU SpMV II

New idea: instead of assigning one thread per row, assign one warp per row.

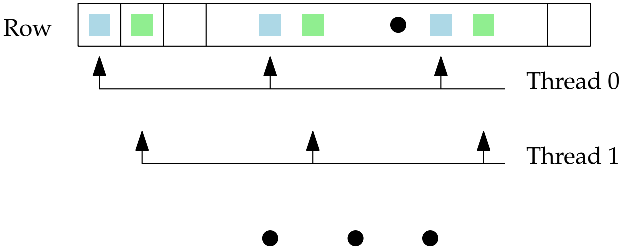
- Each thread in a warp sums up a piece of the dot product
- At end, a parallel reduction combines the pieces of the sum
- Unrolling helps speed the reduction.



GPU SpMV II

New idea: instead of assigning one thread per row, assign one warp per row.

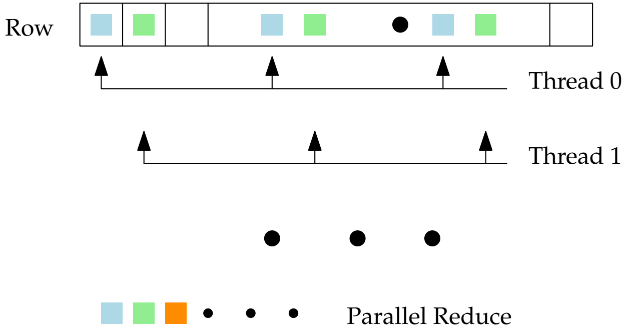
- Each thread in a warp sums up a piece of the dot product
- At end, a parallel reduction combines the pieces of the sum
- Unrolling helps speed the reduction.



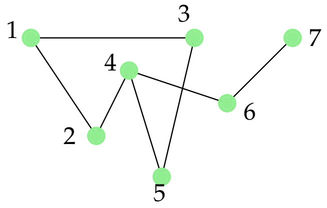
GPU SpMV II

New idea: instead of assigning one thread per row, assign one warp per row.

- Each thread in a warp sums up a piece of the dot product
- At end, a parallel reduction combines the pieces of the sum
- Unrolling helps speed the reduction.

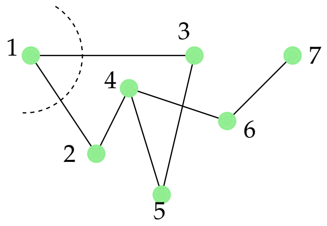


Compute a BFS ordering of a graph[MGG12]



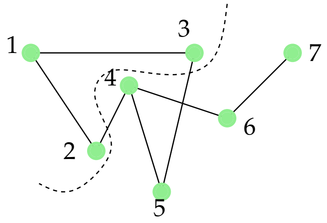
- BFS is notoriously hard to parallelize well
- Main challenge is managing the nonuniform vertex and edge frontier

Compute a BFS ordering of a graph[MGG12]



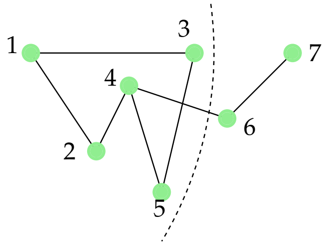
- BFS is notoriously hard to parallelize well
- Main challenge is managing the nonuniform vertex and edge frontier

Compute a BFS ordering of a graph[MGG12]



- BFS is notoriously hard to parallelize well
- Main challenge is managing the nonuniform vertex and edge frontier

Compute a BFS ordering of a graph[MGG12]



- BFS is notoriously hard to parallelize well
- Main challenge is managing the nonuniform vertex and edge frontier

Matrix View

If A is the adjacency matrix of a graph, and x is a vector representing the current vertex frontier, then

$$y = x^\top A$$

is the new frontier.

- This is under the $(\min, +)$ algebra, rather than $(+, \times)$ for regular matrix operations
- Methods from sparse matrix multiplication can be used here.
- But this is very expensive.

Plan:

- Replace matrix view by a “parallel” frontier expansion
- Carefully manage duplicate neighbors

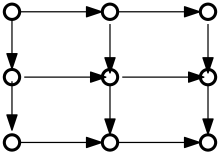
Better Implementation I

How to construct a new frontier from current frontier ?

- Every thread manages one node, and counts its neighbors.
- Once we have all neighbors, we invoke a prefix sum.
- Now threads can write new frontier into shared memory.

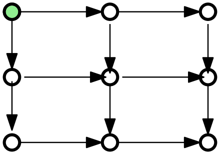
Better Implementation II

New frontier can have many duplicates in it, if individual threads share neighbors.



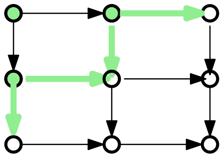
Better Implementation II

New frontier can have many duplicates in it, if individual threads share neighbors.



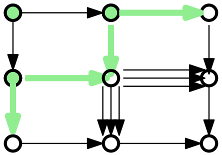
Better Implementation II

New frontier can have many duplicates in it, if individual threads share neighbors.



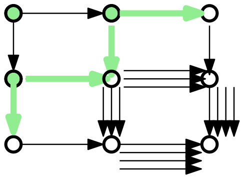
Better Implementation II

New frontier can have many duplicates in it, if individual threads share neighbors.

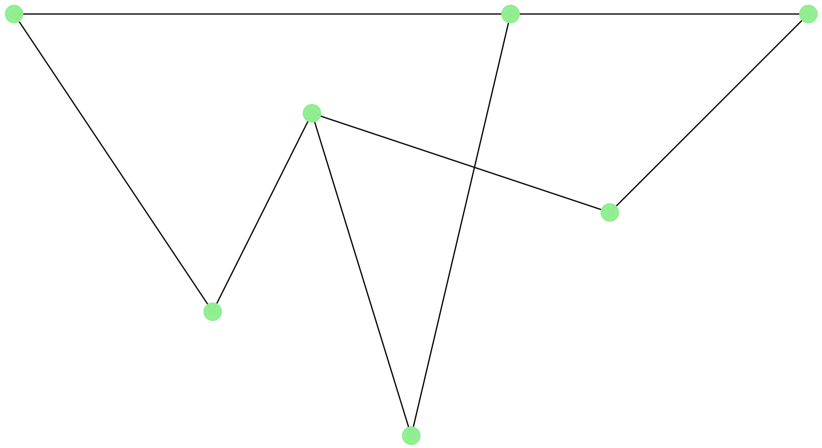


Better Implementation II

New frontier can have many duplicates in it, if individual threads share neighbors.

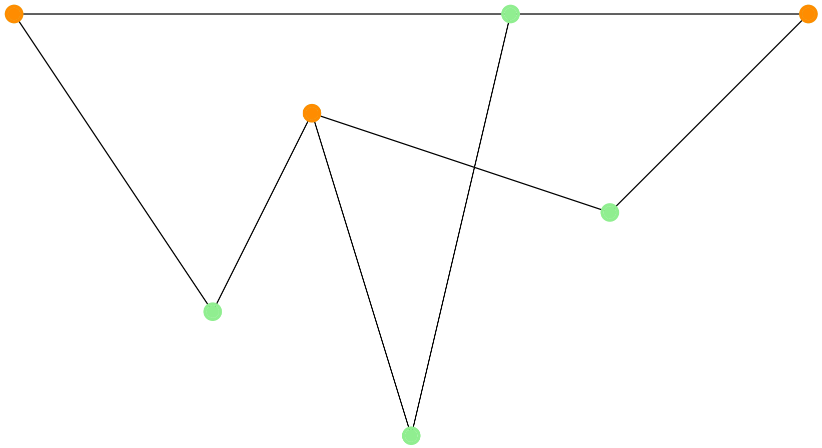


Graph Coloring



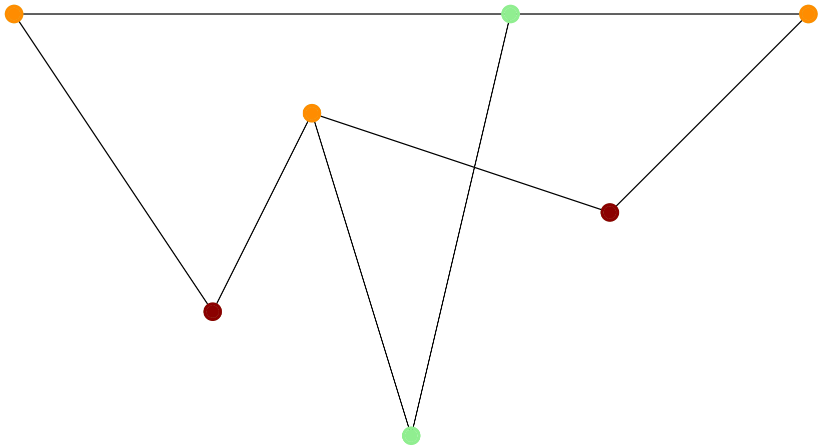
- Core problem in graph optimization
- Register allocation, spectrum assignment, scheduling, ...

Graph Coloring



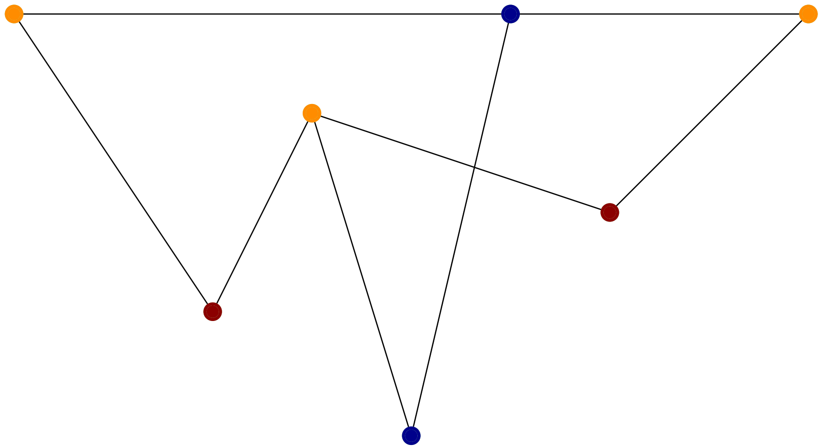
- Core problem in graph optimization
- Register allocation, spectrum assignment, scheduling, ...

Graph Coloring



- Core problem in graph optimization
- Register allocation, spectrum assignment, scheduling, ...

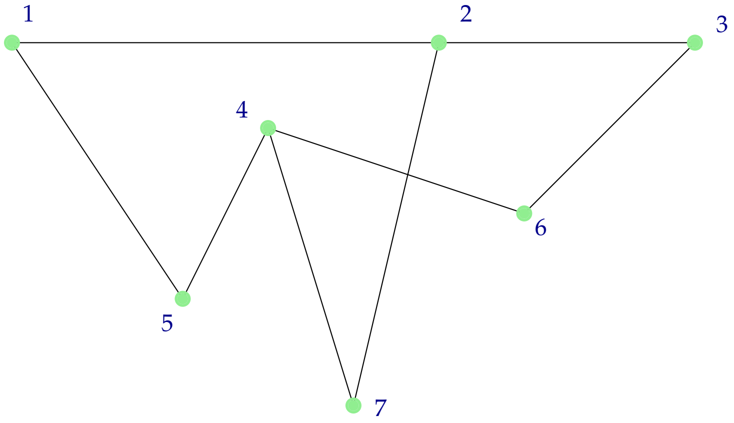
Graph Coloring



- Core problem in graph optimization
- Register allocation, spectrum assignment, scheduling, ...

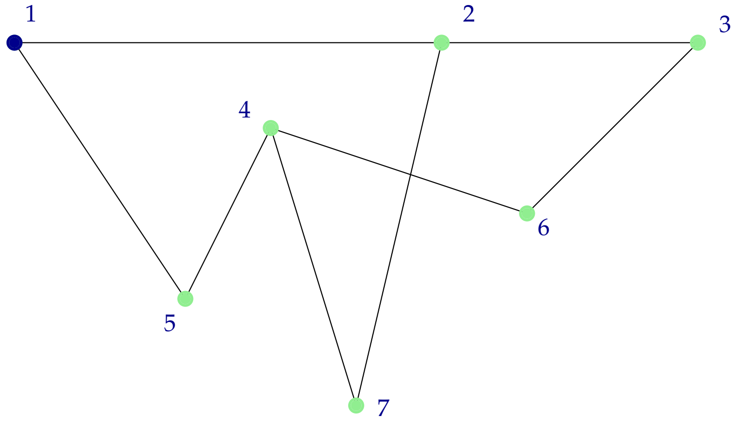
What do we know about it

- NP-hard, and $n^{1-\epsilon}$-hard to approximate
- Many heuristics (based on greedy ordering)
  - Fix an arbitrary ordering of the vertices
  - Color a vertex with the smallest feasible color.



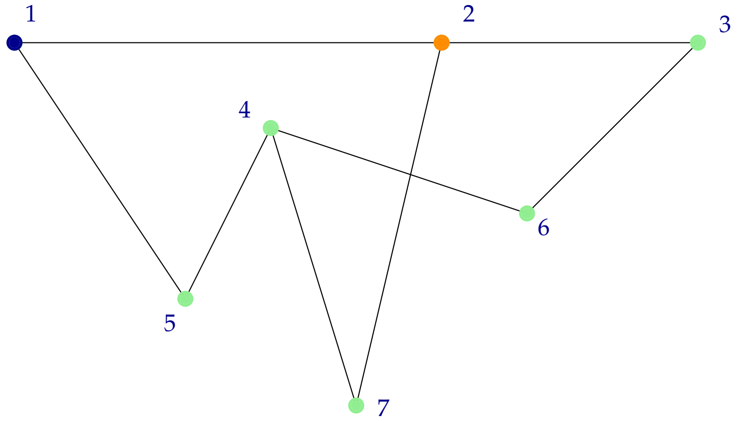
What do we know about it

- NP-hard, and $n^{1-\epsilon}$-hard to approximate
- Many heuristics (based on greedy ordering)
    - Fix an arbitrary ordering of the vertices
    - Color a vertex with the smallest feasible color.



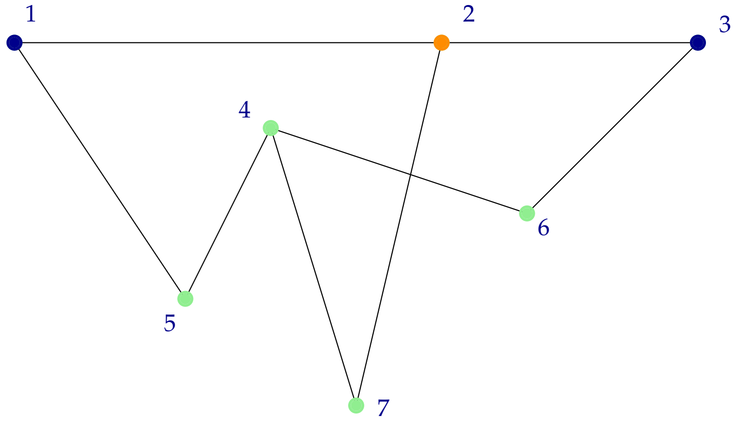
What do we know about it

- NP-hard, and $n^{1-\epsilon}$-hard to approximate
- Many heuristics (based on greedy ordering)
    - Fix an arbitrary ordering of the vertices
    - Color a vertex with the smallest feasible color.



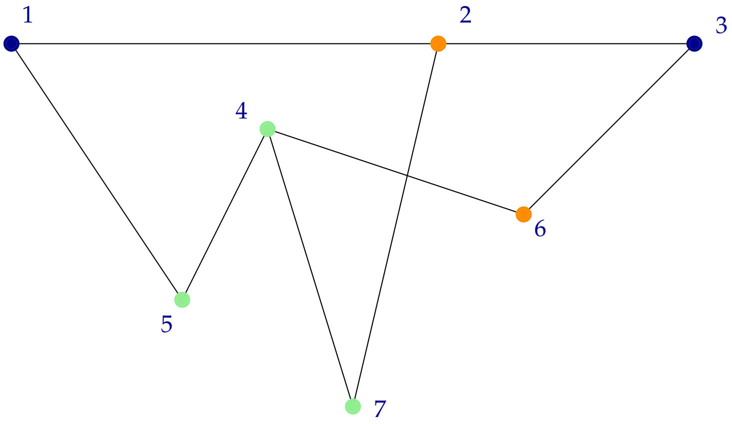
What do we know about it

- NP-hard, and $n^{1-\epsilon}$-hard to approximate
- Many heuristics (based on greedy ordering)
  - Fix an arbitrary ordering of the vertices
  - Color a vertex with the smallest feasible color.



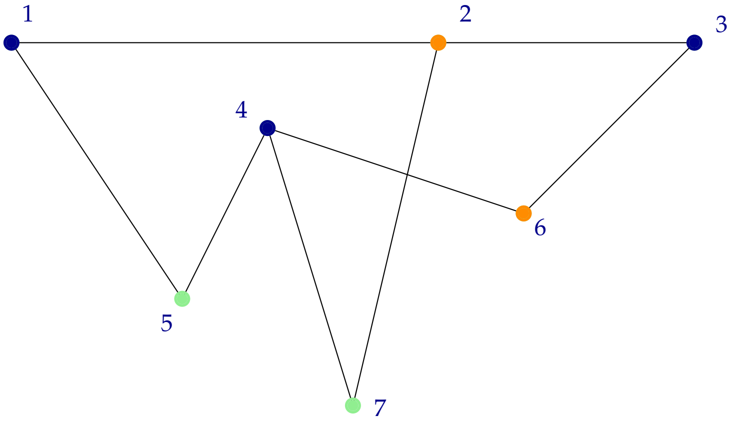
What do we know about it

- NP-hard, and $n^{1-\epsilon}$-hard to approximate
- Many heuristics (based on greedy ordering)
    - Fix an arbitrary ordering of the vertices
    - Color a vertex with the smallest feasible color.



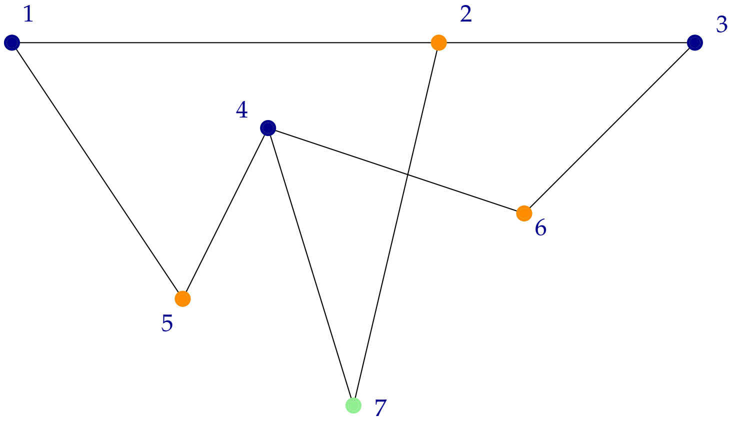
What do we know about it

- NP-hard, and $n^{1-\epsilon}$-hard to approximate
- Many heuristics (based on greedy ordering)
  - Fix an arbitrary ordering of the vertices
  - Color a vertex with the smallest feasible color.



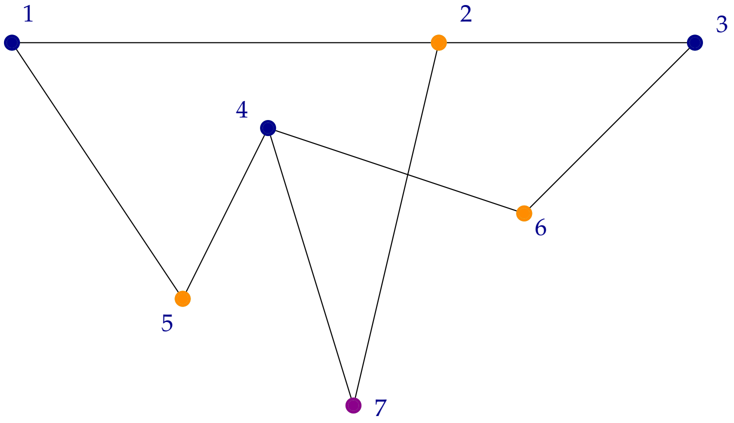
What do we know about it

- NP-hard, and $n^{1-\epsilon}$-hard to approximate
- Many heuristics (based on greedy ordering)
    - Fix an arbitrary ordering of the vertices
    - Color a vertex with the smallest feasible color.



What do we know about it

- NP-hard, and $n^{1-\epsilon}$-hard to approximate
- Many heuristics (based on greedy ordering)
  - Fix an arbitrary ordering of the vertices
  - Color a vertex with the smallest feasible color.



Ordering Heuristics

First Fit	Choose any ordering
SDO/LDO	Color is allocated to vertex with highest “saturation” (number of distinct neighboring colors) and then highest degree.
MAX OUT	Choose vertex that has the maximum number of edges going out of the subgraph
MIN OUT	Choose vertex that has *fewest* number of edges out of the subgraph.

Parallel Coloring

1. Partition the graph into roughly equal-sized pieces that have very few connections between them
2. Color each piece in parallel
3. Fix conflicts at boundaries of pieces

GPU has fine-grained parallelism, faster SIMD processors. Can we do better ?

GPU Coloring[GZL+11]

1. Arbitrarily partition vertices into pieces
2. Color each piece in a thread block, but use common color pool
3. Suppose conflicts occur (at boundary)
    - Erase colors of conflicted nodes
    - Try to color them again
    - Repeat until number of conflicts is small
    - Shift to CPU.

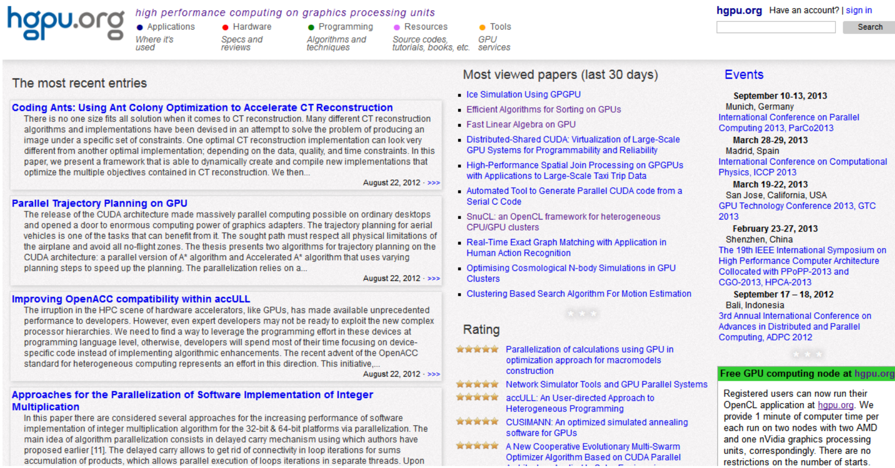
Summary

- GPU SIMD makes conflict checking very easy
- Doing careful partitioning (METIS) doesn’t really help (GPU is more tolerant to “bad” partitioning)
- CPU is very slow to resolve conflicts sequentially: best to use it when number of conflicts is small
- GPU heuristics give good quality colorings (not sure why!)

Software Tools

- CUDPP (`http://code.google.com/p/cudpp` (basic data-parallel tools)
- CUBLAS (`http://developer.nvidia.com/cuda/cublas`)
- Thrust (`https://code.google.com/p/thrust/`) (template library)
- Cusp (`http://code.google.com/p/cusp-library/`) (sparse linear algebra)

Research Tools



hgpu.org *high performance computing on graphics processing units*

- Applications — *Where it's used*
- Hardware — *Specs and reviews*
- Programming — *Algorithms and techniques*
- Resources — *Source codes, tutorials, books, etc.*
- Tools — *GPU services*

hgpu.org Have an account? | sign in

Search

The most recent entries

Coding Ants: Using Ant Colony Optimization to Accelerate CT Reconstruction
There is no one size fits all solution when it comes to CT reconstruction. Many different CT reconstruction algorithms and implementations have been devised in an attempt to solve the problem of producing an image under a specific set of constraints. One optimal CT reconstruction implementation can look very different from another optimal implementation; depending on the data, quality, and time constraints. In this paper, we present a framework that is able to dynamically create and compile new implementations that optimize the multiple objectives contained in CT reconstruction. We then...
August 22, 2012 · >>>

Parallel Trajectory Planning on GPU
The release of the CUDA architecture made massively parallel computing possible on ordinary desktops and opened a door to enormous computing power of graphics adapters. The trajectory planning for aerial vehicles is one of the tasks that can benefit from it. The sought path must respect all physical limitations of the airplane and avoid all no-flight zones. The thesis presents two algorithms for trajectory planning on the CUDA architecture: a parallel version of A* algorithm and Accelerated A* algorithm that uses varying planning steps to speed up the planning. The parallelization relies on a...
August 22, 2012 · >>>

Improving OpenACC compatibility within accULL
The irruption in the HPC scene of hardware accelerators, like GPUs, has made available unprecedented performance to developers. However, even expert developers may not be ready to exploit the new complex processor hierarchies. We need to find a way to leverage the programming effort in these devices at programming language level, otherwise, developers will spend most of their time focusing on device-specific code instead of implementing algorithmic enhancements. The recent advent of the OpenACC standard for heterogeneous computing represents an effort in this direction. This initiative,...
August 22, 2012 · >>>

Approaches for the Parallelization of Software Implementation of Integer Multiplication
In this paper there are considered several approaches for the increasing performance of software implementation of integer multiplication algorithm for the 32-bit & 64-bit platforms via parallelization. The main idea of algorithm parallelization consists in delayed carry mechanism using which authors have proposed earlier [11]. The delayed carry allows to get rid of connectivity in loop iterations for sums accumulation of products, which allows parallel execution of loops iterations in separate threads. Upon

Most viewed papers (last 30 days)

- Ice Simulation Using GPGPU
- Efficient Algorithms for Sorting on GPUs
- Fast Linear Algebra on GPU
- Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability
- High-Performance Spatial Join Processing on GPGPUs with Applications to Large-Scale Taxi Trip Data
- Automated Tool to Generate Parallel CUDA code from a Serial C Code
- SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters
- Real-Time Exact Graph Matching with Application in Human Action Recognition
- Optimising Cosmological N-body Simulations in GPU Clusters
- Clustering Based Search Algorithm For Motion Estimation

Rating

- ★★★★★ Parallelization of calculations using GPU in optimization approach for macromodels construction
- ★★★★★ Network Simulator Tools and GPU Parallel Systems
- ★★★★★ accULL: An User-directed Approach to Heterogeneous Programming
- ★★★★★ CUSIMANN: An optimized simulated annealing software for GPUs
- ★★★★★ A New Cooperative Evolutionary Multi-Swarm Optimizer Algorithm Based on CUDA Parallel

Events

September 10-13, 2013
Munich, Germany
International Conference on Parallel Computing 2013, ParCo2013

March 28-29, 2013
Madrid, Spain
International Conference on Computational Physics, ICCP 2013

March 19-22, 2013
San Jose, California, USA
GPU Technology Conference 2013, GTC 2013

February 23-27, 2013
Shenzhen, China
The 19th IEEE International Symposium on High Performance Computer Architecture Collocated with PPoPP-2013 and CGO-2013, HPCA-2013

September 17 – 18, 2012
Bali, Indonesia
3rd Annual International Conference on Advances in Distributed and Parallel Computing, ADPC 2012

Free GPU computing node at hgpu.org

Registered users can now run their OpenCL application at hgpu.org. We provide 1 minute of computer time per each run on two nodes with two AMD and one nVidia graphics processing units, correspondingly. There are no restrictions on the number of starts.

Overview of lectures

- GPU in the BC era: vertex and fragment shaders. Can do Voronoi diagrams !
- SIMD view key to designing and exploiting behavior of card.
- CUDA provides general purpose SIMD framework
- Low-level SIMD violations can lose many factors in performance
- Parallel reduction and prefix sum is an important primitive.
- Many applications: dense systems, sparse systems, geometry, . . .
- For efficient code, reduce to known primitives like reduction/prefix sm

Debate

The GPU represents the realistic future of high intensity parallel computing. SIMD is the only way to get the throughput needed for many problems, and once memory buses become faster, GPUs will become the primary model.

Versus

While the GPU can demonstrate great performance, the hoops you have to jump through to get this performance are so constraining and so artificial that GPUs will never be more than a boutique processor that is great for games.

Questions?

References I

A.V.P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall.
Evaluating graph coloring on gpus.
In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 297–298. ACM, 2011.

M. Harris, G.E. Blelloch, B.M. Maggs, N.K. Govindaraju, B. Lloyd, W. Wang, M. Lin, D. Manocha, P.K. Smolarkiewicz, L.G. Margolin, et al.
Optimizing parallel reduction in cuda.
Proc. of ACM SIGMOD, 21, 13:104–110, 2007.

N. Leischner, V. Osipov, and P. Sanders.
Gpu sample sort.
In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. Ieee, 2010.

Duane Merrill, Michael Garland, and Andrew Grimshaw.
Scalable gpu graph traversal.
In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

References II

J. Nickolls, I. Buck, M. Garland, and K. Skadron.
Scalable parallel programming with cuda.
Queue, 6(2):40–53, 2008.

G. Rong and T.S. Tan.
Jump flooding in gpu with applications to voronoi diagram and distance transform.
In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116. ACM, 2006.

S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens.
Scan primitives for gpu computing.
In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106. Eurographics Association, 2007.

D.P.R. Srikanth, K. Kothapalli, R. Govindarajulu, and PJ Narayanan.
Parallelizing two dimensional convex hull on nvidia gpu and cell be.
In *International Conference on High Performance Computing (HiPC)*, pages 1–5, 2009.

References III

M. Zechner and M. Granitzer.
Accelerating k-means on the graphics processor via cuda.
In *Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on*, pages 7–15. IEEE, 2009.