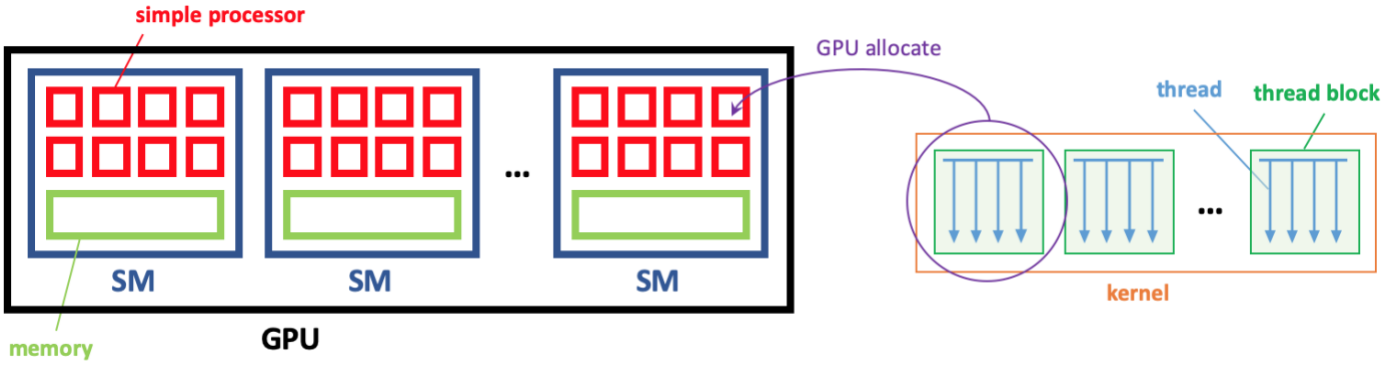
Parallel Computing Model

CPU vs. GPU

	CPU	GPU
Control Hardware	One; complex; flexibility	Mass; simple; restrictive
Power	High	Low for each unit
Optimization Objective	Latency (s)	Throughput (#task/s)
Tasks Good At		Many repetitive tasks

GPU Architecture

- A GPU has many streaming multiprocessors (SMs)
- An SM has many simple processors that can run many parallel threads
- A thread block contains many threads; all threads in one block cooperate to solve a subproblem
- A kernel contains many blocks; no communication between blocks



- The programmer defines blocks in software; the GPU allocates blocks to SMs
- Scalability: SM can schedule another block immediately when it's idle

Guarantee:

- An SM may run more than one block; a block can be run in only one SM
- All blocks in a kernel finish before any block from the next kernel

Not guarantee:

- The order of threads and blocks
- When and where blocks will run

Memory

- Local memory: accessed by the thread only
- Shared memory: accessed by threads in a thread block (`__shared__`)
- Global memory: accessed by any threads
- Host memory: memory in CPU, copying from/to GPU global memory

Speed: local > shared >> global >> host

- CPU is "host"; dominate (send command to) the GPU "device"
- Both CPU & GPU have their own separated memories (DRAM)
- CPU & GPU are connected by PCIe (commonly 6GB/s)
- PCI can transfer memory that has been page locked or pinned; it keeps a special chunk of pinned host memory set aside for this purpose

Synchronization

Barrier

Point in the program where threads stop and wait (`__syncthreads`)
When all threads have reached the barrier, they can proceed
Implicit barriers between kernels

Atomic Operation

Use atomic functions (e.g., `atomicAdd`)
Only certain operations & data types supported
Serialize access to memory - slow

CUDA Programming Pipeline

Code: xxx.cu
Compiler: nvcc

1. Allocate GPU memory (`cudaMalloc`)
2. Copy data from CPU to GPU (`cudaMemcpy`)
3. Launch kernels on GPU to compute (`kernel_name<<<dim3,dim3>>>(param)`)
4. Kernel get index (`threadIdx.x`)
   Then computes tasks sequentially (`__global__ void kernel_name(param)`)
5. Copy data from GPU to CPU

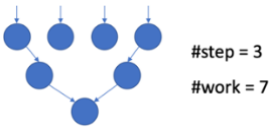
Dynamic Parallel

Launch another kernel inside a kernel
Works well for recursive parallelism (e.g., quicksort)
Resources except global memory are not shared between parent kernel and child kernel
Launching on-the-fly makes it greater GPU utilization

Algorithm

Fundamental

Step: how long to complete a particular computation task for a thread
Work: Total amount of tasks for a kernel



Reduce

Tasks like sum an array
Reduce elements into a value with reduction operator $\otimes$ (binary & associative)

$$\text{Reduce}(\{S_1, S_2\}, \otimes) = \text{Reduce}(S_1, \otimes) \otimes \text{Reduce}(S_2, \otimes)$$

Step complexity: $\log n$

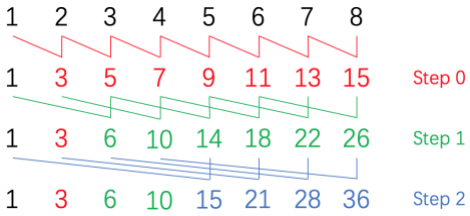
Scan

Tasks like prefix sum
Exclusive scan: $\text{Scan}([a_1, a_2 \dots, a_n], \otimes) = [\emptyset, a_1, a_1 \otimes a_2, \dots, a_1 \otimes \dots \otimes a_{n-1}]$
Inclusive scan: $\text{Scan}([a_1, a_2 \dots, a_n], \otimes) = [a_1, a_1 \otimes a_2, \dots, a_1 \otimes \dots \otimes a_n]$
Identity element $\emptyset$: $x \otimes \emptyset = x$

Segmented Scan

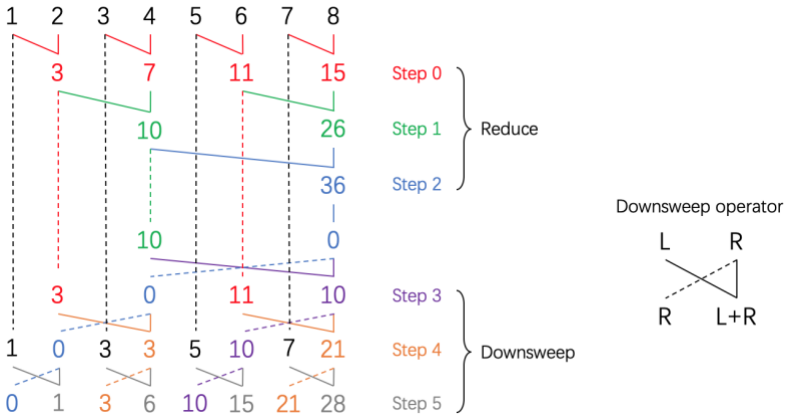
To scan many small arrays, instead of scan them individually, combine them into a large array and scan
Use another 0/1 array to mark the heads of small arrays

Hillis-Steele Inclusive Scan



Step efficiency: $\log n$
Work complexity: $n \log n$
At step i, for element x, add itself to the element in $x - 2^i$

Blelloch Exclusive Scan



Step complexity: $2 \log n$
Work efficiency: $O(n)$

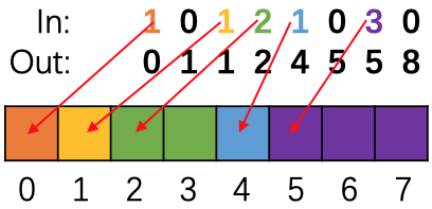
Histogram

Tasks like divide data into bins
Set #bin local bins in each thread, and do accumulation, then reduce #thread local bins to a global bin
Avoid using atomic operations to global bins

Compact & Allocate

Compact: tasks like filter a subset of elements
Allocate: tasks like allocate dynamic space for element with different sizes

1. Run predicate to get an array of prediction (Compact: true/false; Allocate: #space)
2. Exclusive sum scan, output is scatter addresses for new array
3. Scatter elements of the prediction array into the new array using its address



Sparse Matrix / Dense Vector Multiplication (SpMv)

CSR (compressed sparse row format):

$$\begin{bmatrix} a & & b \\ c & d & e \\ & & f \end{bmatrix}$$

```
Value   [a b c d e f]
Column  [0 2 0 1 2 2]
RowPtr  [0 2 5]
```

Multiplication:

1. `S`: Create a segmented representation of values
2. `V`: Gather vector values using column
3. `M`: Pairwise multiply using `S` and `V`
4. `O`: Inclusive segmented sum scan on `M`

$$\begin{bmatrix} a & & b \\ c & d & e \\ & & f \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

```
S = [a   b  | c   d   e  | f ]
V = [x   z    x   y   z    z ]
M = [ax  bz | cx  dy  ez | fz]
O = [ax+bz cx+dy+ez fz]
```

Odd-Even Sort (Brick Sort)

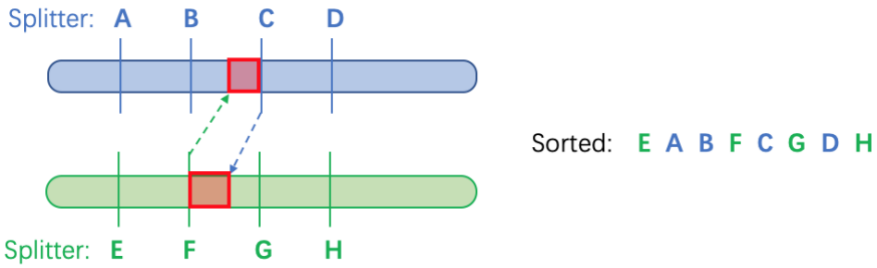
Sort pairs of elements alternatively

Step complexity: n

Work complexity: n^2

Merge Sort

- [*Stage* 1] tasks are small, #task >> #SMs
  Complete small merge tasks per thread using a serial algorithm
- [*Stage* 2] tasks are medium, #task ≈ #SMs
  Complete a task per thread block using **parallel merge**
  1. Start merge two sorted list A and B
  2. Binary search the position p of Ai in B, get the final address i+p in merged list for Ai
  3. Every thread belongs to one element in A or B, run parallelly in shared memory
- [*Stage* 3] tasks are large, #task << #SMs
  1. Break up two big lists with splitters with a medium span
  2. Sort splitters using parallel merge
  3. For two adjacent splitters from different lists, sort their confusing parts using parallel merge
  4. Get all addresses of elements and construct the final merged list

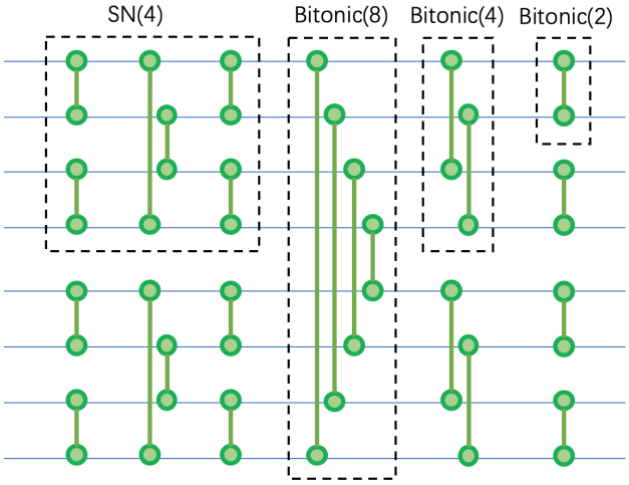


Sorting Network

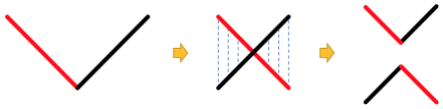
Obviousness: behavior is independent of programs or data

Bitonic Sorting Network

Step complexity: $O(\log^2 n)$

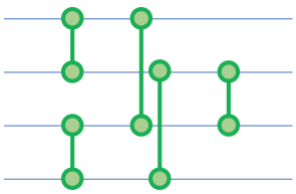


Bitonic sequence: a sequence only change direction at most once
A bitonic sequence can be separated into two bitonic subsequences (a larger half & a smaller half) by pairwise comparison



Bitonic sorting network works like divide and conquer (note: sequence reverse in bitonic separation)

Odd-Even Merge Sorting Network



Radix Sort

1. Sort by bits from low to high
2. Split elements into two sets based on i^{th} bit, otherwise preserve order
   Using two passes of compact algorithm to get new addresses of elements with 0s or 1s
3. Move to the next bit & repeat

Step complexity: $O(\log^2 n)$

Quicksort

1. Choose pivot element
2. Compare all elements with pivot, split into three subarrays: <P, =P or >P
3. Segmented compact on the array to get new addresses
4. Recurse on each subarray

Optimization

High-Level Strategies

- Maximize arithmetic intensity:

$$\frac{\text{math}}{\text{memory}}$$

  - Maximize compute operations per thread
  - Minimize time spend on memory per thread
    Move frequently-accessed data to fast memory
  - Coalesced global memory access
    GPU is most efficient when thread read/write contiguous memory locations
- Avoid thread divergence
  Make threads in the same thread block execute in the same branch or loop number

Little's Law

#byte delivered = average latency of each transaction * bandwidth
GPU: #useful bytes delivered = average latency * bandwidth

Occupancy

Find the bottleneck of the device, and control the parameters to optimize
Each SM has a limited resource (by command `deviceQuery`):

- Thread blocks: 8
- Threads: 1536/2048
- Maximum #thread per block: 1024
- Registers for all threads: 65536
- Bytes of shared memory: 16K-48K

Parallel Optimization Strategies

Data Layout Transformation

Recognize data layout for better memory performance: AoS or SoA

```
struct S {
    float a;
    float b;
} A[8];
```

Array of structures (AoS)

```
struct S {
    float a[8];
    float b[8];
} A;
```

Structure of arrays (SoA)

Scatter-To-Gather Transformation

Scatter: `b[i-1] += a[i]/3;  b[i] += a[i]/3;  b[i+1] += a[i]/3;`
Gather: `b[i] = (a[i-1] + a[i] + a[i+1]) / 3;`
Gather is better than scatter because:

- Scatter causes many potential conflicting writes
- Gather has many overlapping reads

Tiling

Buffer data into fast on-chip storage for repeated access (shared memory, caches)
Process large N×N matrix by small P×P tiles to advance coalesced reads/writes

Privatization

Set thread-local or thread-block-local output buffers, and reduce local buffers to get global output
Avoid write conflicts in global memory

Binning / Spatial Data Structure

Build data structure that maps output locations to a subset of the relevant input data (e.g., grid partition on a geometry map, only consider adjacent grids)

Compaction

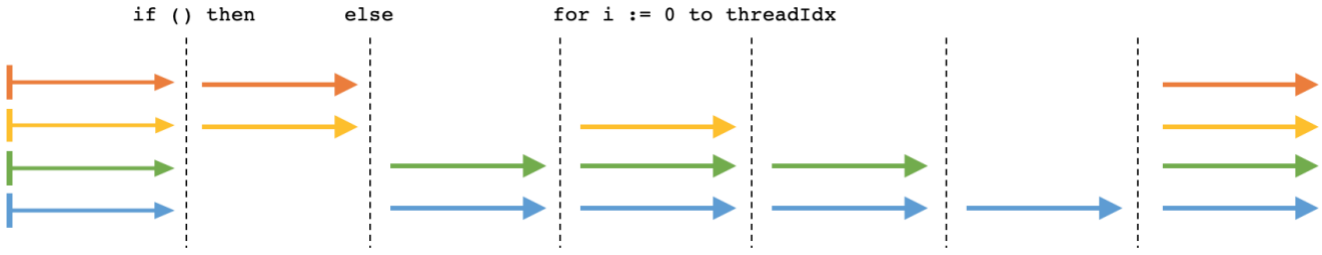
Compact: if only process a few elements in the original array, compact them into a new array

Regularization

Reorganize input data (e.g., order) to reduce workload imbalance (irregular parallelism => regular parallelism)
Different element, different kernel/algorithm/resource

Thread Divergence

Threads in one thread block execute in the same flow
Branches (`if`) and loops (`for`) will make threads idle when executing different paths



Warp

Set of threads (e.g., #thread=32) that execute the same instruction at a time
At most 32x slowdown when executing no matter how many threads in a thread block
CUDA assigns thread IDs to warps from fastest to slowest: x > y > z
Should consider whether adjacent thread likely to take different paths
SIMD: Single Instruction, Multiple Data
Modern CPU use vector instructions set (with vector registers): SSE (e.g., float4) or AVX (e.g., float8)
SIMT: Single Instruction, Multiple Thread

CPU-GPU Interaction

- Use `cudaHostMalloc` to allocate pinned host memory
- Use `cudaHostRegister` to take some data that are already allocated and pin it
- Use `cudaMemcpyAsync` to copy asynchronously

Streams

Overlap memory transfer & compute by putting them in different streams and executing together
Help fill GPU with smaller kernels

```
cudaStream_t s1;
cudaStreamCreate(&s1);
cudaMemcpy(dst, src, numBytes, s1);
kernel_name<<<dim3, dim3, s1>>>(params);
cudaStreamDestroy(s1);
```

Assorted Math Optimization

- Use double precision only when you need it
- Use intrinsics when possible (e.g., `__sin`, `__cos`, `__exp`)

Application

All Pairs N-Body

Tasks like simulate particle system
Calculate forces on each element by interactive forces with other N-1 elements, then move each element based on the force

1. Divide N×N interaction matrix into P×P tiles to use shared memory
2. In each P×P tile, launch P threads for each row instead of launch P×P threads for each cell

SpMV

Sparse Matrix Multiply Vector

- Launch a thread for each row in sparse matrix: row length diverse, not efficient
- Launch a thread for each element in sparse matrix: too many threads
- Hybrid: launch a thread for the first x elements of each row, then launch threads for the rest elements

BFS

Brute Force (Bellman-Ford-Like)

1. Iterate on depths
2. For each depth d, parallel launch a thread for each edge, update the depth of vertices with depth d+1
3. Maintain a bool flag for stop

Step complexity: $O(V)$
Work complexity: $O(VE)$

Quadratic GPU Implementation

1. Store the edges using forward star (an array of edges sorted by vertices + an array of start locations for vertices)
2. Iterate on depths
3. For each depth d, maintain a list of frontiers (vertices with the depth d)
4. Fetch frontiers' neighbors from edge array: Allocate
5. Remove vertices that are already visited from the neighbors: Compact

Step complexity: $O(V)$
Work complexity: $O(V^2)$

List Ranking

Input: each node knows its successor & the starting node
Output: all node in order

1. Find 2^kth successors for all node
   The $2x$th successor of node u can be find by its xth successor: $\text{succ}[u, 2x] = \text{succ}[\text{succ}[u, x], x]$
   Step complexity: $\log n$
   Work complexity: $n \log n$
2. Construct all node in order
   From the starting node, use first x nodes to find first $2x$ nodes using $\text{succ}[u, x]$
   Step complexity: $\log n$

Cuckoo Hashing

Typical hash table need linked chains, which is not good for parallel (workload imbalance)

1. Set multiple hash tables (with different hash functions); each value in each table stores one element
2. Construct: find table t; if occupied, find table t+1
   If the element x cannot occupy in all tables, kick out the element y already in table 1 and let y to find another table (augmenting path)
   Not guarantee to arrange all elements, often set an iteration upper limit
3. Find: find all tables parallel
   Step complexity: const.

Tools

CUDA Libraries

cuBLAS:	Implementation of Basic Linear Algebra Subroutines
cuFFT:	Fast Fourier transform, 1D/2D/3D FFT routines
cuSPARSE:	BLAS-like routines for sparse matrix formats
cuRAND:	Pseudo- and quasi-random generation routines (from particular distribution)
NPP:	NVIDIA Performance Primitives, low-level image processing primitives
Magma:	GPU + multicore CPU LAPACK routines
CULA:	Eigensolvers, matrix factorization & solvers
ArrayFire:	Framework for data-parallel manipulation of array data

Thrust

Analogous to C++ Standard Template Library (STL)
Host-side code/interface running on GPU without kernel (e.g., `thrust::device_vector`)

- Including containers, iterators, borrow niceties from Boost library
- Sort, scan, reduce, reduce-by-key, transform input vectors,
- Interoperate with CUDA code

CUDA Unbound (CUB)

Software reuse in CUDA kernels, enabling unbounded parameters like #thread, shared memory
Enable optimization auto-tuning
CUDA DMA: focus movement of data from global into shared memory; make it easier to use shared memory

Other Languages

PyCUDA:	Warp CUDA C++
Copperhead:	data-parallel subset of Python (generate Thrust code)
CUDA Fortran:	Fortran with CUDA constructs
Halide:	Image processing DSL (domain specific language)
MATLAB:	Both warp CUDA kernels and directly targeting using only MATLAB functions; with profiler

Cross-Platform Solutions

OpenCL: Similar to CUDA
OpenGL Compute: Similar to CUDA, integrated with OpenGL graphics
OpenACC: Directives-based approach, add a few codes to legacy codes, complier automatically parallelize

nSight

NVIDIA tool to measure bandwidth utilization
NVIDIA Visual Profiler (NVVP)
Linux/Mac: nSight Eclipse
Windows: nSight Visual Studio