Faculty of Mathematics and Information Science

WARSAW UNIVERSITY OF TECHNOLOGY

Graphic Processors in Computational Applications

Part 3 – Algorithms

dr inż. Krzysztof Kaczmarski

2021

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Politechnika Warszawska

Unia Europejska
Europejski Fundusz Społeczny

Materiały sponsorowane przez:

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Fundusze Europejskie Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Politechnika Warszawska

Unia Europejska Europejski Fundusz Społeczny

Goals for today:

- Get familiar with parallel algorithms building blocks
- Understand several interesting algorithms

Part 3 – Algorithms

Taxonomy of parallel machines

Introduction

RAM – Random Access Machine

PRAM – Parallel Random Access Machine (EREW, CREW, ERCW, CRCW)

- E{R,W} – Exclusive read/write – two processors cannot access the same memory address in the same time
- C{R,W} – Concurrent read/write

Taxonomy of parallel machines

Introduction

RAM – Random Access Machine

PRAM – Parallel Random Access Machine
(EREW, CREW, ERCW, CRCW)

E{R,W} – Exclusive read/write – two processors cannot access the same memory address in the same time

C{R,W} – Concurrent read/write

It is also important to know if execution of all commands is synchronized or not.

- in case of GPU (CUDA) we may assure synchronization only within a block of threads.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001

Taxonomy of parallel machines

RAM – Random Access Machine

PRAM – Parallel Random Access Machine (EREW, CREW, ERCW, CRCW)

- E{R,W} – Exclusive read/write – two processors cannot access the same memory address in the same time
- C{R,W} – Concurrent read/write

It is also important to know if execution of all commands is synchronized or not.

- in case of GPU (CUDA) we may assure synchronization only within a block of threads.
- this property may spoil algorithms and needs additional work

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001

Taxonomy of parallel machines

RAM – Random Access Machine

PRAM – Parallel Random Access Machine
(EREW, CREW, ERCW, CRCW)

E{R,W} – Exclusive read/write – two processors cannot access the same memory address in the same time

C{R,W} – Concurrent read/write

It is also important to know if execution of all commands is synchronized or not.

- in case of GPU (CUDA) we may assure synchronization only within a block of threads.
- this property may spoil algorithms and needs additional work
- in several cases it is enough to separate input and output (see array reverse example)

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001

Speedup

T – time, W – work, N – number of processors,
$*_s$ – before improvement (sequential),
$*_p$ – after improvement (parallel)

$$S_T(N) = \frac{T_s}{T_p}$$

Parallelization of Sequential Code

Introduction

Speedup

T – time, W – work, N – number of processors,
$*_s$ – before improvement (sequential),
$*_p$ – after improvement (parallel)

$$S_T(N) = \frac{T_s}{T_p}$$

$$S_W(N) = \frac{W_p}{W_s}$$

Parallelization of Sequential Code

Amdahl's Law

Constant Problem Size: $W_p = W_s$

T – time, P – fraction of parallelized program,
N – number of processors

$$T_p(N) \;\; = \;\; (1 - P)\, T_s + P \frac{T_s}{N}$$

Parallelization of Sequential Code

Amdahl's Law

Constant Problem Size: $W_p = W_s$

T – time, P – fraction of parallelized program,
N – number of processors

$$
\begin{aligned}
T_p(N) &= (1-P)\,T_s + P\,\frac{T_s}{N} \\
S_T(N) &= \frac{T_s}{T_p(N)} = \frac{T_s}{(1-P)\,T_s + P\,\frac{T_s}{N}}
\end{aligned}
$$

Parallelization of Sequential Code

Amdahl's Law

Constant Problem Size: $W_p = W_s$

T – time, P – fraction of parallelized program,
N – number of processors

$$
\begin{aligned}
T_p(N) &= (1-P)\,T_s + P\,\frac{T_s}{N} \\
S_T(N) &= \frac{T_s}{T_p(N)} = \frac{T_s}{(1-P)\,T_s + P\,\frac{T_s}{N}} \\
S_T(N) &= \frac{1}{(1-P) + \frac{P}{N}}
\end{aligned}
$$

Parallelization of Sequential Code

Amdahl's Law – examples

- $P = \frac{1}{2}, N = 2 \rightarrow S = \frac{1}{(1-\frac{1}{2})+\frac{\frac{1}{2}}{2}} = 1.25$
- $P = 1 \rightarrow S = N$
- $P = \frac{1}{2}, N = 20 \rightarrow S = \frac{1}{(1-\frac{1}{2})+\frac{\frac{1}{2}}{20}} \approx 1.904$

If N is large then we can omit $\frac{P}{N}$:

- $P = \frac{3}{4} \rightarrow S = \frac{1}{(1-\frac{3}{4})} = 4$
- $P = \frac{1}{6} \rightarrow S = \frac{1}{(1-\frac{1}{6})} = \frac{6}{5} = 1.2$

Parallelization of Sequential Code

Amdahl's Law

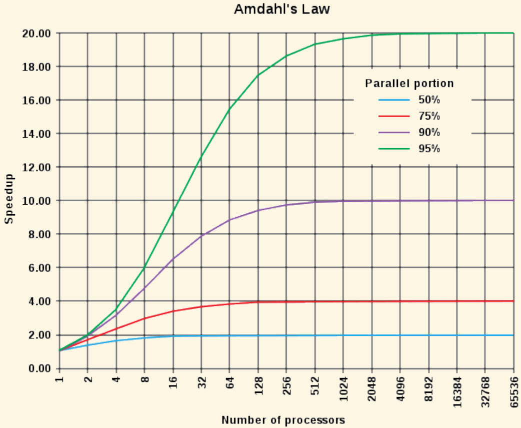


Figure: Speedup limits by Amhdl's Law

Constant Total Computation Time: $T_s = T_p$

T – time, P – portion of parallel program time,
N – Number of processors

$$W_s = (1 - P)W_s + P \cdot W_s$$

Parallelization of Sequential Code

Gustafson's Law

Constant Total Computation Time: $T_s = T_p$

T – time, P – portion of parallel program time,
N – Number of processors

$$
\begin{aligned}
W_s &= (1-P)W_s + P \cdot W_s \\
W_p(N) &= (1-P)W_s + N \cdot P \cdot W_s
\end{aligned}
$$

Parallelization of Sequential Code

Gustafson's Law

Constant Total Computation Time: $T_s = T_p$

T – time, P – portion of parallel program time,
N – Number of processors

$$
\begin{aligned}
W_s &= (1-P)W_s + P \cdot W_s \\
W_p(N) &= (1-P)W_s + N \cdot P \cdot W_s \\
S_W(N) &= \frac{W_p(N)}{W_s} = \frac{(1-P)W_s + N \cdot P \cdot W_s}{W_s}
\end{aligned}
$$

Parallelization of Sequential Code

Gustafson's Law

Constant Total Computation Time: $T_s{=}T_p$

T – time, P – portion of parallel program time,
N – Number of processors

$$
\begin{aligned}
W_s &= (1-P)W_s + P \cdot W_s \\
W_p(N) &= (1-P)W_s + N \cdot P \cdot W_s \\
S_W(N) &= \frac{W_p(N)}{W_s} = \frac{(1-P)W_s + N \cdot P \cdot W_s}{W_s} \\
S_W(N) &= 1 - P + N \cdot P
\end{aligned}
$$

Parallelization of Sequential Code

Gustafson's Law

Constant Total Computation Time: $T_s = T_p$

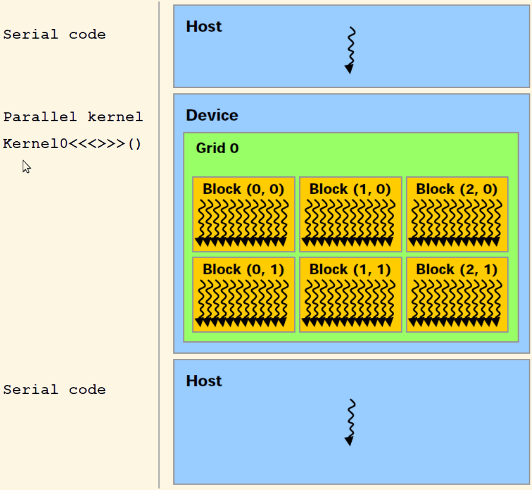
T – time, P – portion of parallel program time,
N – Number of processors

$$
\begin{aligned}
W_s &= (1-P)W_s + P \cdot W_s \\
W_p(N) &= (1-P)W_s + N \cdot P \cdot W_s \\
S_W(N) &= \frac{W_p(N)}{W_s} = \frac{(1-P)W_s + N \cdot P \cdot W_s}{W_s} \\
S_W(N) &= 1 - P + N \cdot P
\end{aligned}
$$

- $P = \frac{1}{2}, N = 2 \rightarrow S = 1 - \frac{1}{2} + 2 \cdot \frac{1}{2} = 1.5$
- $P = \frac{1}{2}, N = 20 \rightarrow S = 1 - \frac{1}{2} + 20 \cdot \frac{1}{2} = 10.5$

Heterogeneous programming with host and device

Introduction



NVIDIA. Cuda c++ programming guide. www.nvidia.com/cuda

Part 3 – Algorithms

Scatter/Gather Operations

Introduction

Parallel threads may easily access any location in global or shared memory with two possible behaviors:

Scatter/Gather Operations

Introduction

Parallel threads may easily access any location in global or shared memory with two possible behaviors:

Gather

Single thread reads from many locations writes to one. Can accumulate data in private registers. Possible shared memory utilization while reading.

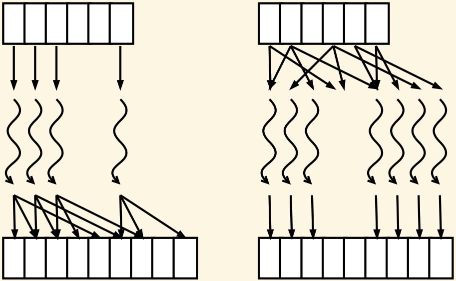
Scatter

Single thread reads from one location writes to many.
Scatter leads to possible write conflicts:

- use atomic writes (slow down)
- change to gather if possible
- privatization (more memory)

Examples of scatter and gather

Introduction



scatter: electrons-protons one thread per particle, naive histogram

gather: electrons-protons one thread per output pixel, matrix multiplication, fish simulation one thread per a fish

Part 3 – Algorithms

Map

Introduction

Definition (Map)

The map operation takes a function F (well defined for given input values) and an array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the array

$$[F(x_0), F(x_1), \ldots, F(x_{n-1})].$$

- This task is one of *embarrassingly parallel* problems.

Definition (Map)

The map operation takes a function F (well defined for given input values) and an array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the array

$$[F(x_0), F(x_1), \ldots, F(x_{n-1})].$$

- This task is one of *embarrassingly parallel* problems.
- One of possible optimizations – $\mathsf{map}(F) \circ \mathsf{map}(G) = \mathsf{map}(F \circ G)$

Map

Introduction

Definition (Map)

The map operation takes a function F (well defined for given input values) and an array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the array

$$[F(x_0), F(x_1), \ldots, F(x_{n-1})].$$

- This task is one of *embarrassingly parallel* problems.
- One of possible optimizations –
  $\mathsf{map}(F) \circ \mathsf{map}(G) = \mathsf{map}(F \circ G)$
- Also an idea for loops parallelism
  (if subsequent iterations are independent).

Definition (Map)

The map operation takes a function F (well defined for given input values) and an array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the array

$$[F(x_0), F(x_1), \ldots, F(x_{n-1})].$$

- This task is one of *embarrassingly parallel* problems.
- One of possible optimizations –
  $\mathrm{map}(F) \circ \mathrm{map}(G) = \mathrm{map}(F \circ G)$
- Also an idea for loops parallelism
  (if subsequent iterations are independent).
- In CUDA F must be defined as `__device__` function.

Map

Introduction

Definition (Map)

The map operation takes a function F (well defined for given input values) and an array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the array

$$[F(x_0), F(x_1), \ldots, F(x_{n-1})].$$

- This task is one of *embarrassingly parallel* problems.
- One of possible optimizations – $\mathsf{map}(F) \circ \mathsf{map}(G) = \mathsf{map}(F \circ G)$
- Also an idea for loops parallelism (if subsequent iterations are independent).
- In CUDA F must be defined as `__device__` function.
- CUDA supports 2d and 3d arrays of threads .

Map

Introduction

Definition (Map)

The map operation takes a function F (well defined for given input values) and an array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the array

$$[F(x_0), F(x_1), \ldots, F(x_{n-1})].$$

- This task is one of *embarrassingly parallel* problems.
- One of possible optimizations – $\mathsf{map}(F) \circ \mathsf{map}(G) = \mathsf{map}(F \circ G)$
- Also an idea for loops parallelism (if subsequent iterations are independent).
- In CUDA F must be defined as `__device__` function.
- CUDA supports 2d and 3d arrays of threads .
- ... more dimensions must be simulated.

Part 3 – Algorithms

Prefix sums

Introduction

Definition (Scan – Array all-prefix-sums)

The scan operation takes a binary associative operator $\oplus$, and an array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the array

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \cdots \oplus x_{n-1})].$$

Prefix sums

Introduction

Definition (Scan – Array all-prefix-sums)

The scan operation takes a binary associative operator $\oplus$, and an array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the array

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \cdots \oplus x_{n-1})].$$

Definition (Prescan)

The prescan operation takes a binary associative operator $\oplus$ with identity I, and an array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the array

$$[I, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \cdots \oplus x_{n-2})].$$

Guy E Blelloch. Prefix sums and their applications, 1990

Scan – naive solution

Introduction

```
for d := 1 to log₂ n do
   forall k in parallel do
      if k ⩾ 2^d then x[k] := x[k − 2^(d−1)] + x[k]
```

0	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	$\sum_0^0 x_i$	$\sum_0^1 x_i$	$\sum_1^2 x_i$	$\sum_2^3 x_i$	$\sum_3^4 x_i$	$\sum_4^5 x_i$	$\sum_5^6 x_i$	$\sum_6^7 x_i$
2	$\sum_0^0 x_i$	$\sum_0^1 x_i$	$\sum_0^2 x_i$	$\sum_0^3 x_i$	$\sum_1^4 x_i$	$\sum_2^5 x_i$	$\sum_3^6 x_i$	$\sum_4^7 x_i$
3	$\sum_0^0 x_i$	$\sum_0^1 x_i$	$\sum_0^2 x_i$	$\sum_0^3 x_i$	$\sum_0^4 x_i$	$\sum_0^5 x_i$	$\sum_0^6 x_i$	$\sum_0^7 x_i$

Not work-efficient: $O(n \log(n))$ compared to sequential $O(n)$

W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986

Scan – work-efficient solution (I)

Introduction

Up-sweep (reduce) phase (scan)

```
for d := 0 to log_2 n − 1 do
   for k from 0 to n − 1 by 2^d + 1 in parallel do
      x[k + 2^(d+1) − 1] := x[k + 2^d − 1] + x [k + 2^(d+1) − 1]
```

3	x_0	$\sum_0^1 x_i$	x_2	$\sum_0^3 x_i$	x_4	$\sum_4^5 x_i$	x_6	$\sum_0^7 x_i$
2	x_0	$\sum_0^1 x_i$	x_2	$\sum_0^3 x_i$	x_4	$\sum_4^5 x_i$	x_6	$\sum_4^7 x_i$
1	x_0	$\sum_0^1 x_i$	x_2	$\sum_2^3 x_i$	x_4	$\sum_4^5 x_i$	x_6	$\sum_6^7 x_i$
	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Guy E Blelloch. Prefix sums and their applications, 1990

Scan – work-efficient solution (II)

Introduction

Down-sweep (reduce) phase (prescan)

```
x[n − 1] := 0
for d := log₂ n down to 0 do
   for k from 0 to n − 1 by 2^(d+1) in parallel do
      t := x[k + 2^d − 1]
      x[k + 2^d − 1] := x [k + 2^(d+1) − 1]
      x[k + 2^(d+1) − 1] := t + x [k + 2^(d+1) − 1]
```

1	x_0	$\sum_0^1 x_i$	x_2	$\sum_0^3 x_i$	x_4	$\sum_4^5 x_i$	x_6	0
2	x_0	$\sum_0^1 x_i$	x_2	0	x_4	$\sum_4^5 x_i$	x_6	$\sum_0^3 x_i$
3	x_0	0	x_2	$\sum_0^1 x_i$	x_4	$\sum_0^3 x_i$	x_6	$\sum_0^5 x_i$
4	0	x_0	$\sum_0^1 x_i$	$\sum_0^2 x_i$	$\sum_0^3 x_i$	$\sum_0^4 x_i$	$\sum_0^5 x_i$	$\sum_0^6 x_i$

Guy E Blelloch. Prefix sums and their applications, 1990

Scan – work-efficient solution (III)

Introduction

- Work-efficient $O(n)$

Scan – work-efficient solution (III)

- Work-efficient $O(n)$
- Prescan result may be converted to scan by: $\text{scan}[i] = \text{prescan}[i] \oplus x_i$ or by shifting the result by one element left and adding the last element of prescan to the last element of the original input.

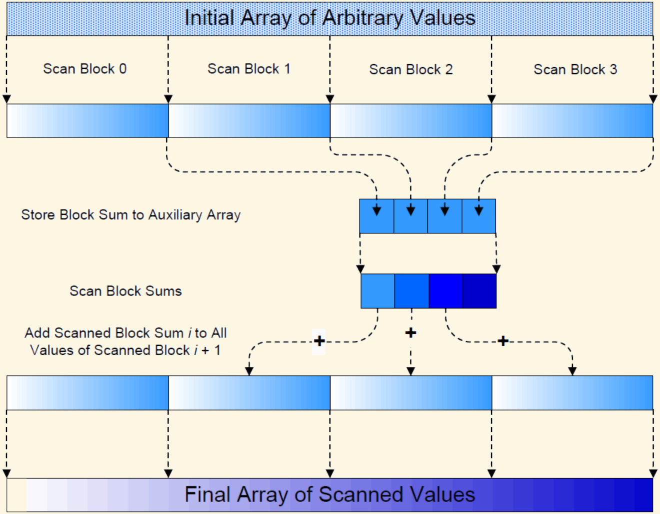
Scan – work-efficient solution (III)

- Work-efficient $O(n)$
- Prescan result may be converted to scan by: $\text{scan}[i] = \text{prescan}[i] \oplus x_i$ or by shifting the result by one element left and adding the last element of prescan to the last element of the original input.
- Additional care for bigger arrays since blocks of threads must be synchronized.

Part 3 – Algorithms

Scan of arbitrary size arrays

Introduction



Mark Harris. Parallel prefix sum (scan) with CUDA. www.nvidia.com/cuda, 2007

Part 3 – Algorithms

Applications of prefix sums algorithm

Sample applications of scan

- Computation of minimum, maximum, average, etc. of an array
- Lexical comparison of strings of characters
- Addition of multi-precision numbers that cannot be represented in a single machine word
- Evaluation of polynomials
- Solving of recurrence equations
- Radix sort
- Quick sort
- Solving tridiagonal linear systems
- Removal of marked elements from an array
- Dynamical allocation of processors
- Lexical analysis (parsing into tokens)
- Searching for regular expressions
- Implementation of some tree operations

Pack operation

Sample applications of scan

Definition (Pack)

For given array of input values A and flags array F (true/false), pack returns array with elements from A array which are marked as true in flags array only.

Pack operation

Sample applications of scan

Definition (Pack)

For given array of input values A and flags array F (true/false), pack returns array with elements from A array which are marked as true in flags array only.

Definition (Enumerate)

For given input vector of true/false flags F enumerate returns vector containing at each position a number of predeceasing true values in F.

Pack operation

Sample applications of scan

Definition (Pack)

For given array of input values A and flags array F (true/false), pack returns array with elements from A array which are marked as true in flags array only.

Definition (Enumerate)

For given input vector of true/false flags F enumerate returns vector containing at each position a number of predeceasing true values in F.

Example:

A	6	3	4	8	1	2	4	2
F	0	0	0	1	1	0	0	1

Pack operation

Sample applications of scan

Definition (Pack)

For given array of input values A and flags array F (true/false), pack returns array with elements from A array which are marked as true in flags array only.

Definition (Enumerate)

For given input vector of true/false flags F enumerate returns vector containing at each position a number of predeceasing true values in F.

Example:

A	6	3	4	8	1	2	4	2
F	0	0	0	1	1	0	0	1
prescan(F)	0	0	0	0	1	2	2	2

Pack operation

Sample applications of scan

Definition (Pack)

For given array of input values A and flags array F (true/false), pack returns array with elements from A array which are marked as true in flags array only.

Definition (Enumerate)

For given input vector of true/false flags F enumerate returns vector containing at each position a number of predeceasing true values in F.

Example:

A	6	3	4	8	1	2	4	2
F	0	0	0	1	1	0	0	1
prescan(F)	0	0	0	0	1	2	2	2
pack(A,F)	8	1	2					

Radix sort

Sample applications of scan

```
procedure split_radix_sort(A, number_of_bits)
  for i from 0 to (number_of_bits - 1)
    A := split(A, A<i>)
```

A	5	7	3	1	4	2	7	2

Radix sort

Sample applications of scan

```
procedure split_radix_sort(A, number_of_bits)
  for i from 0 to (number_of_bits - 1)
    A := split(A, A<i>)
```

A	5	7	3	1	4	2	7	2
A<0>	1	1	1	1	0	0	1	0

Radix sort

Sample applications of scan

```
procedure split_radix_sort(A, number_of_bits)
  for i from 0 to (number_of_bits - 1)
    A := split(A, A<i>)
```

A	5	7	3	1	4	2	7	2

A<0>	1	1	1	1	0	0	1	0
split(A, A<0>)	4	2	2	5	7	3	1	7

Radix sort

Sample applications of scan

```
procedure split_radix_sort(A, number_of_bits)
  for i from 0 to (number_of_bits - 1)
    A := split(A, A<i>)
```

A	5	7	3	1	4	2	7	2

A<0>	1	1	1	1	0	0	1	0
split(A, A<0>)	4	2	2	5	7	3	1	7

A<1>	0	1	1	0	1	1	0	1

Radix sort

Sample applications of scan

```
procedure split_radix_sort(A, number_of_bits)
  for i from 0 to (number_of_bits - 1)
    A := split(A, A<i>)
```

A	5	7	3	1	4	2	7	2
A<0>	1	1	1	1	0	0	1	0
split(A, A<0>)	4	2	2	5	7	3	1	7
A<1>	0	1	1	0	1	1	0	1
split(A, A<1>)	4	5	1	2	2	7	3	7

Radix sort

Sample applications of scan

```
procedure split_radix_sort(A, number_of_bits)
  for i from 0 to (number_of_bits - 1)
    A := split(A, A<i>)
```

A	5	7	3	1	4	2	7	2
A<0>	1	1	1	1	0	0	1	0
split(A, A<0>)	4	2	2	5	7	3	1	7
A<1>	0	1	1	0	1	1	0	1
split(A, A<1>)	4	5	1	2	2	7	3	7
A<2>	1	1	0	0	0	1	0	1

Radix sort

Sample applications of scan

```
procedure split_radix_sort(A, number_of_bits)
  for i from 0 to (number_of_bits - 1)
    A := split(A, A<i>)
```

A	5	7	3	1	4	2	7	2
A<0>	1	1	1	1	0	0	1	0
split(A, A<0>)	4	2	2	5	7	3	1	7
A<1>	0	1	1	0	1	1	0	1
split(A, A<1>)	4	5	1	2	2	7	3	7
A<2>	1	1	0	0	0	1	0	1
split(A, A<2>)	1	2	2	3	4	5	7	7

Radix sort

Sample applications of scan

```
procedure split_radix_sort(A, number_of_bits)
  for i from 0 to (number_of_bits - 1)
    A := split(A, A<i>)
```

A	5	7	3	1	4	2	7	2
A<0>	1	1	1	1	0	0	1	0
split(A, A<0>)	4	2	2	5	7	3	1	7
A<1>	0	1	1	0	1	1	0	1
split(A, A<1>)	4	5	1	2	2	7	3	7
A<2>	1	1	0	0	0	1	0	1
split(A, A<2>)	1	2	2	3	4	5	7	7

Guy E Blelloch. Prefix sums and their applications, 1990

Split with scan

Sample applications of scan

```
procedure split(A, Flags)
   I_down := sum_prescan(not(Flags))
   I_up := n - sum_scan(reverse_order(Flags))
   forall i in parallel do
      if (Flags[i])
         Index[i] := I_up[i]
      else
         Index[i] := I_down[i]
   result := permute(A, Index)
```

A	5	7	3	1	4	2	7	2
Flags	1	1	1	1	0	0	1	0
I_down	0	0	0	0	0	1	2	2
I_up	3	4	5	6	6	6	7	7
Index	3	4	5	6	0	1	7	2
permute(A, Index)	4	2	2	5	7	3	1	7

Guy E Blelloch. Prefix sums and their applications, 1990

Segmented scan

Sample applications of scan

Guy E Blelloch. Prefix sums and their applications, 1990

Definition (Segmented scan)

For given array of input values $[a_0, \ldots, a_{n-1}]$ and array of flags $[f_0, \ldots, f_{n-1}]$ segmented scan returns array $[x_0, \ldots, x_{n-1}]$ satisfying the equation:

$$x_i = \begin{cases} a_0 & i = 0 \\ \begin{cases} a_i & f_i = 1 \\ (x_{i-1} \oplus a_i) & f_i = 0 \end{cases} & 0 < i < n \end{cases}$$

Segmented scan

Sample applications of scan

Guy E Blelloch. Prefix sums and their applications, 1990

Definition (Segmented scan)

For given array of input values $[a_0, \dots, a_{n-1}]$ and array of flags $[f_0, \dots, f_{n-1}]$ segmented scan returns array $[x_0, \dots, x_{n-1}]$ satisfying the equation:

$$x_i = \begin{cases} a_0 & i = 0 \\ \begin{cases} a_i & f_i = 1 \\ (x_{i-1} \oplus a_i) & f_i = 0 \end{cases} & 0 < i < n \end{cases}$$

- Original scan may be segmented in such a way that the scan starts again at each segment boundary.

Segmented scan

Sample applications of scan

Guy E Blelloch. Prefix sums and their applications, 1990

Definition (Segmented scan)

For given array of input values $[a_0, \ldots, a_{n-1}]$ and array of flags $[f_0, \ldots, f_{n-1}]$ segmented scan returns array $[x_0, \ldots, x_{n-1}]$ satisfying the equation:

$$x_i = \begin{cases} a_0 & i = 0 \\ \begin{cases} a_i & f_i = 1 \\ (x_{i-1} \oplus a_i) & f_i = 0 \end{cases} & 0 < i < n \end{cases}$$

- Original scan may be segmented in such a way that the scan starts again at each segment boundary.
- Implementation of this method is much slower than original unsegmented scan.

Example of segmented scan (Up-sweep phase)

Sample applications of scan

```
for d = 1 to log_2 n - 1 do
    for k = 0 to n - 1 by 2^(d+1) in parallel do
        if f[k + 2^(d+1) - 1] = false then
            x[k + 2^(d+1) - 1] := x[k + 2^d - 1] + x[k + 2^(d+1) - 1]
        f[k + 2^(d+1) - 1] := f[k + 2^d - 1] | f[k + 2^(d+1) - 1]
```

f	1	0	0	1	0	0	0	0
x	x_0	$\sum_0^1 x_i$	x_2	x_3	x_4	$\sum_4^5 x_i$	x_6	$\sum_3^7 x_i$
x	x_0	$\sum_0^1 x_i$	x_2	x_3	x_4	$\sum_4^5 x_i$	x_6	$\sum_4^7 x_i$
x	x_0	$\sum_0^1 x_i$	x_2	x_3	x_4	$\sum_4^5 x_i$	x_6	$\sum_6^7 x_i$
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In Mark Segal and Timo Aila, editors, *Graphics Hardware*, pages 97–106. Eurographics Association, 2007

Example of segmented scan (Down-sweep phase)

Sample applications of scan

```
x[n-1] := 0
for d := log_2 n - 1 down to 0 do
   for k from 0 to n - 1 by 2^(d+1) in parallel do
      t := x[k + 2^d - 1]
      x[k + 2^d - 1] := x [k + 2^(d+1) - 1]
      if f[k + 2^d] = true then
         x[k + 2^(d+1) - 1] := 0
      else if f[k + 2^d - 1] = true then
         x[k + 2^(d+1) - 1] := t
      else
         x[k + 2^(d+1) - 1] := t + x [k + 2^(d+1) - 1]
      f[k + 2^d - 1] := false
```

f	1	0	0	1	0	0	0	0
x	x_0	$\sum_0^1 x_i$	x_2	x_3	x_4	$\sum_4^5 x_i$	x_6	0
x	x_0	$\sum_0^1 x_i$	x_2	0	x_4	$\sum_4^5 x_i$	x_6	x_3
f	1	0	0	0	0	0	0	0
x	x_0	0	x_2	$\sum_0^1 x_i$	x_4	x_3	x_6	$\sum_3^5 x_i$
x	0	x_0	$\sum_0^1 x_i$	$\sum_0^2 x_i$	x_3	$\sum_3^4 x_i$	$\sum_3^5 x_i$	$\sum_3^6 x_i$

Parallel Quicksort

Sample applications of scan

```
procedure quicksort(keys)
   seg_flags[0] := 1
   while not_sorted(keys)
      pivots := seg_copy(keys, seg_flags)
      f := pivots <=> keys
      keys := seg_split(keys, f, seg_flags)
      seg_flags := new_seg_flags(keys, pivots, seg_flags)
```

Parallel Quicksort

Sample applications of scan

```
procedure quicksort(keys)
   seg_flags[0] := 1
   while not_sorted(keys)
      pivots := seg_copy(keys, seg_flags)
      f := pivots <=> keys
      keys := seg_split(keys, f, seg_flags)
      seg_flags := new_seg_flags(keys, pivots, seg_flags)
```

key	6.4	9.2	3.4	1.6	8.7	4.1	9.2	3.4
seg_flags	1	0	0	0	0	0	0	0

Parallel Quicksort

Sample applications of scan

```
procedure quicksort(keys)
   seg_flags[0] := 1
   while not_sorted(keys)
      pivots := seg_copy(keys, seg_flags)
      f := pivots <=> keys
      keys := seg_split(keys, f, seg_flags)
      seg_flags := new_seg_flags(keys, pivots, seg_flags)
```

key	6.4	9.2	3.4	1.6	8.7	4.1	9.2	3.4
seg_flags	1	0	0	0	0	0	0	0
pivots	6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4
f	$=$	$>$	$<$	$<$	$>$	$<$	$>$	$<$
key:=split(key, f)	3.4	1.6	4.1	3.4	6.4	9.2	8.7	9.2
seg_flags	1	0	0	0	1	1	0	0

Parallel Quicksort

Sample applications of scan

```
procedure quicksort(keys)
   seg_flags[0] := 1
   while not_sorted(keys)
      pivots := seg_copy(keys, seg_flags)
      f := pivots <=> keys
      keys := seg_split(keys, f, seg_flags)
      seg_flags := new_seg_flags(keys, pivots, seg_flags)
```

key	6.4	9.2	3.4	1.6	8.7	4.1	9.2	3.4
seg_flags	1	0	0	0	0	0	0	0
pivots	6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4
f	=	>	<	<	>	<	>	<
key:=split(key, f)	3.4	1.6	4.1	3.4	6.4	9.2	8.7	9.2
seg_flags	1	0	0	0	1	1	0	0
pivots	3.4	3.4	3.4	3.4	6.4	9.2	9.2	9.2
f	=	<	>	=	=	=	<	=
key:=split(key, f)	1.6	3.4	3.4	4.1	6.4	8.7	9.2	9.2
seg_flags	1	1	0	1	1	1	1	0

Guy E Blelloch. Prefix sums and their applications, 1990

Quicksort notes

Sample applications of scan

- Assures equal load for all processors.

Quicksort notes

Sample applications of scan

- Assures equal load for all processors.
- However rises many implementation problems:

Quicksort notes

Sample applications of scan

- Assures equal load for all processors.
- However rises many implementation problems:
  - segmented scan is much (4 times) slower than normal scan

Quicksort notes

Sample applications of scan

- Assures equal load for all processors.
- However rises many implementation problems:
  - segmented scan is much (4 times) slower than normal scan
  - flags vector must be stored with additional care

Quicksort notes

Sample applications of scan

- Assures equal load for all processors.
- However rises many implementation problems:
  - segmented scan is much (4 times) slower than normal scan
  - flags vector must be stored with additional care
- Theoretical time complexity: $O(\frac{n}{p}\log_2 n + \log_2^2 n)$

Part 3 – Algorithms

Part 3 – Algorithms

Definition (Comparator)

Comparator is a device with two inputs (x and y) and two outputs (x' and y') calculating in time $O(1)$ the following function:

$$x' = \min(x, y)$$
$$y' = \max(x, y)$$

Comparator may calculate results only if both input values are available.

Definition (Comparator)

Comparator is a device with two inputs (x and y) and two outputs (x' and y') calculating in time $O(1)$ the following function:

$$x' = \min(x, y)$$
$$y' = \max(x, y)$$

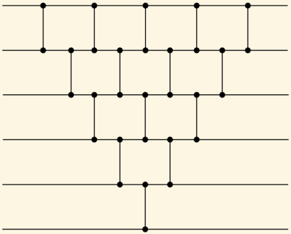
Comparator may calculate results only if both input values are available.

Definition (Sorting network)

Sorting network contains n inputs $a_1, \ldots, a_n$ and n outputs $b_1, \ldots, b_n$. For any given input vector, the output vector is sorted ($b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$). Data flow inside the network has no circles.

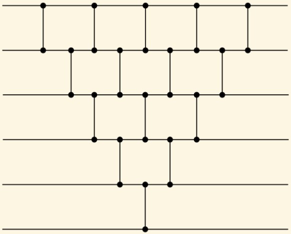
Sorting networks

Sorting networks



Sorting networks can be compared by number of elements or depth.

- Odd-even sorting network – depth: $O(n)$, comparators: $O(n^2)$

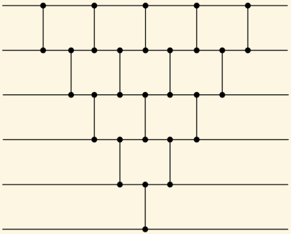


Sorting networks can be compared by number of elements or depth.

- Odd-even sorting network – depth: $O(n)$, comparators: $O(n^2)$
- Merger, bitonic and shell sorting network – depth: $O(\log^2 n)$, comparators: $O(n \log^2 n)$

Sorting networks

Sorting networks



Sorting networks can be compared by number of elements or depth.

- Odd-even sorting network – depth: $O(n)$, comparators: $O(n^2)$
- Merger, bitonic and shell sorting network – depth: $O(\log^2 n)$, comparators: $O(n \log^2 n)$
- Optimal AKS network – depth: $O(\log n)$, comparators: $O(n \log n)$ (impractical)

Theorem (Zero-one principle)

If a comparison network with n inputs sorts all 2^n possible sequences of 0's and 1's correctly, then it sorts all sequences of arbitrary numbers correctly.

Odd-even sort kernel example

Sorting networks

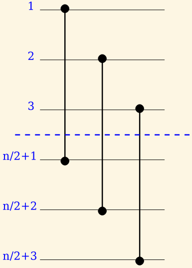
```
__global__ static void sort_shared_mem(float *g_idata, int num_elements)
{
   extern __shared__ float temp[];
   uint thid = threadIdx.x;
   uint m = thid, n = m + 1, off = 0;

   temp[thid] = g_idata[thid];
   __syncthreads();

   if ((m & 1) == 0)
      for (unsigned int i=0; i<num_elements; ++i)
      {
         if ( n <= (num_elements-1) )
            if (temp[m] > temp[n])
               swap( temp[m], temp[n] );
         off = off xor 1;
         m = thid + off;
         n = m + 1;
         __syncthreads();
      }
   __syncthreads();
    g_idata[thid] = temp[thid];
}
```

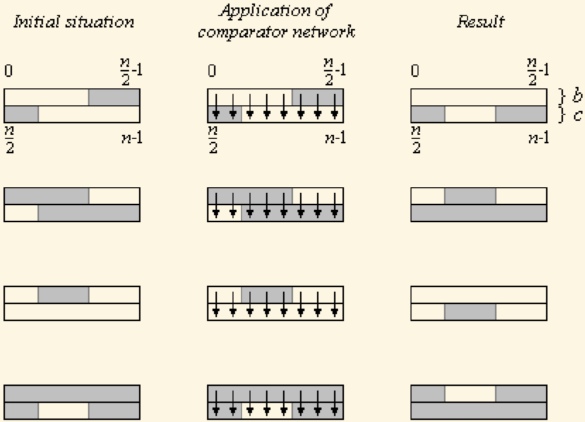
Part 3 – Algorithms

Half-Cleaner[n] network

Sorting networks

Half-Cleaner: input – bitonic,
output – one bitonic, one bitonic-clean.





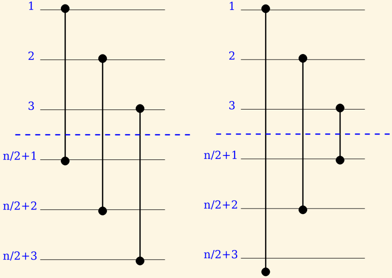
T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001

Donald Knuth. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley, 1973

Half-Cleaner[n] and Merger[n] networks

Sorting networks

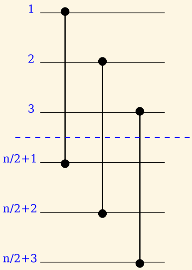
Merger: input – two sorted,
output – two bitonic, one clean.

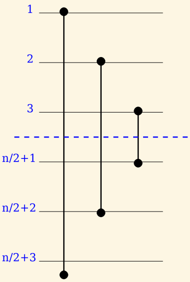


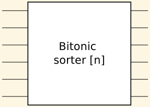
Half-Cleaner[n] and Merger[n] networks

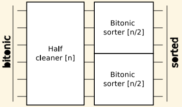
Sorting networks

Merger: input – two sorted,
output – two bitonic, one clean.





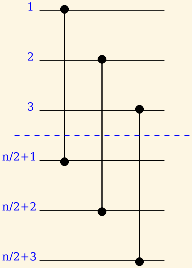


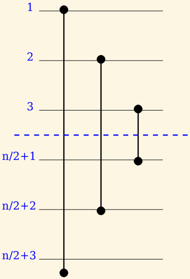


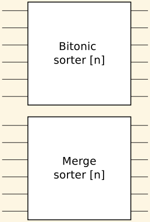
Half-Cleaner[n] and Merger[n] networks

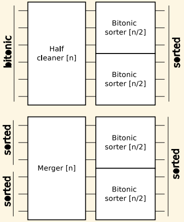
Sorting networks

Merger: input – two sorted,
output – two bitonic, one clean.





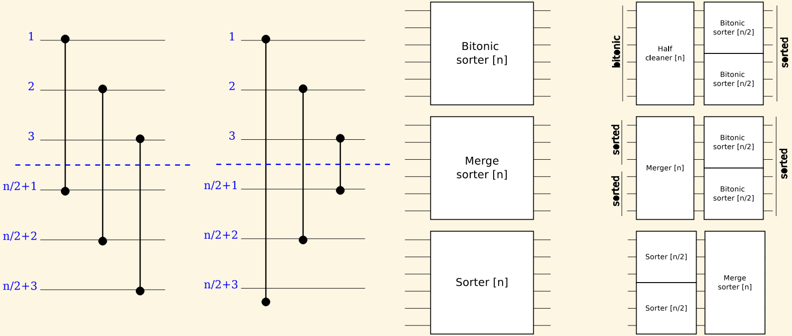




Half-Cleaner[n] and Merger[n] networks

Sorting networks

Merger: input – two sorted,
output – two bitonic, one clean.



T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001

Donald Knuth. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley, 1973

Parallel implementation of bitonic sort

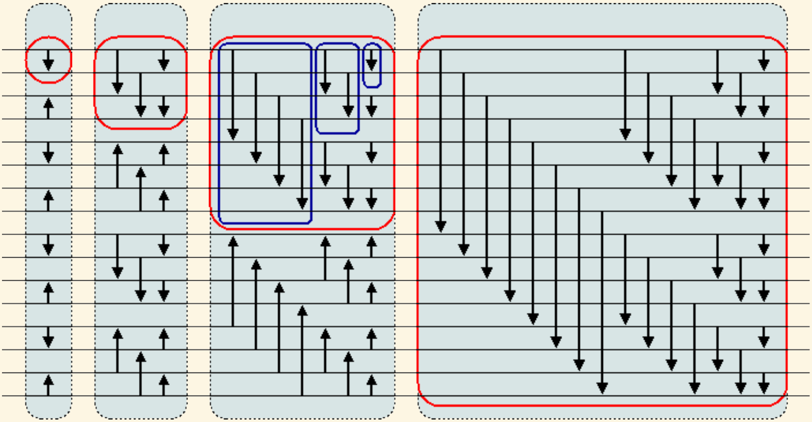
Sorting networks

Nvidia cuda sdk. www.nvidia.com/cuda

```
__global__ static void bitonicSort(int * values)
{
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;
    shared[tid] = values[tid];
    __syncthreads();
    for (unsigned int k = 2; k <= NUM; k *= 2)
        for (unsigned int j = k / 2; j>0; j /= 2)
        {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if ((tid & k) == 0){
                    if (shared[tid] > shared[ixj])
                        swap(shared[tid], shared[ixj]);
                }
                else {
                    if (shared[tid] < shared[ixj])
                        swap(shared[tid], shared[ixj]);
                }
            }
            __syncthreads();
        }
    values[tid] = shared[tid];
}
```

Bitonic sort network

Sorting networks



Part 3 – Algorithms

Part 3 – Algorithms

Interaction of particles

Physical Simulations

1. Integration – Calculate particle properties

Interaction of particles

Physical Simulations

1. Integration – Calculate particle properties
2. Update helper structures – Create grid

Interaction of particles

Physical Simulations

1. Integration – Calculate particle properties
2. Update helper structures – Create grid
3. Process interactions – Calculate collisions

Interaction of particles

Physical Simulations

1. Integration – Calculate particle properties
2. Update helper structures – Create grid
3. Process interactions – Calculate collisions

Interaction of particles

Physical Simulations

1. Integration – Calculate particle properties
2. Update helper structures – Create grid
3. Process interactions – Calculate collisions

Ad. 1. Relatively simple task – forces influence velocity, velocity updates position.

Interaction of particles

Physical Simulations

1. Integration – Calculate particle properties
2. Update helper structures – Create grid
3. Process interactions – Calculate collisions

Ad. 1. Relatively simple task – forces influence velocity, velocity updates position.

Ad. 3. There are generally three types of interactions:

- no interaction – each particle is independent and can be simulated in parallel

Interaction of particles

Physical Simulations

1. Integration – Calculate particle properties
2. Update helper structures – Create grid
3. Process interactions – Calculate collisions

Ad. 1. Relatively simple task – forces influence velocity, velocity updates position.

Ad. 3. There are generally three types of interactions:

- no interaction – each particle is independent and can be simulated in parallel
- unlimited interaction – when all particles influence all other (gravitation)

Interaction of particles

1. Integration – Calculate particle properties
2. Update helper structures – Create grid
3. Process interactions – Calculate collisions

Ad. 1. Relatively simple task – forces influence velocity, velocity updates position.

Ad. 3. There are generally three types of interactions:

- no interaction – each particle is independent and can be simulated in parallel
- unlimited interaction – when all particles influence all other (gravitation)
- interaction limited in distance – when force (or influence) drops off with distance

Interaction of particles

1. Integration – Calculate particle properties
2. Update helper structures – Create grid
3. Process interactions – Calculate collisions

Ad. 1. Relatively simple task – forces influence velocity, velocity updates position.

Ad. 3. There are generally three types of interactions:

- no interaction – each particle is independent and can be simulated in parallel
- unlimited interaction – when all particles influence all other (gravitation)
- interaction limited in distance – when force (or influence) drops off with distance
  - spatial subdivision improves performance – uniform grids

Creating uniform grid of particles in space

- Grid subdivides space into uniformly sized cells

Creating uniform grid of particles in space

- Grid subdivides space into uniformly sized cells
- A single cell contains indices of contained particles (according to particle's center)

Creating uniform grid of particles in space

- Grid subdivides space into uniformly sized cells
- A single cell contains indices of contained particles (according to particle's center)
- We set one thread for each particle

Creating uniform grid of particles in space

- Grid subdivides space into uniformly sized cells
- A single cell contains indices of contained particles (according to particle's center)
- We set one thread for each particle
- However we get conflicts if more particles fall into the same cell

Creating uniform grid of particles in space

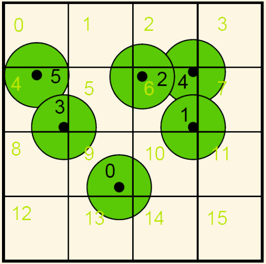
Physical Simulations

- Grid subdivides space into uniformly sized cells
- A single cell contains indices of contained particles (according to particle's center)
- We set one thread for each particle
- However we get conflicts if more particles fall into the same cell

Creating uniform grid of particles in space

Physical Simulations

- Grid subdivides space into uniformly sized cells
- A single cell contains indices of contained particles (according to particle's center)
- We set one thread for each particle
- However we get conflicts if more particles fall into the same cell



Cell index	Count	Particle ID
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

Creating grid with atomic operations

Physical Simulations

Simon Green. CUDA particles. www.nvidia.com/cuda, 2008

```
forall k in parallel do
    j := calcCellNo(k)
    p := gridCounters[j]
    gridCounters[j]++
    gridCells[j][p] := k
```

Notes:

- `gridCells` and `gridCounters` are in global memory.

Creating grid with atomic operations

Physical Simulations

Simon Green. CUDA particles. www.nvidia.com/cuda, 2008

```
forall k in parallel do
    j := calcCellNo(k)
    p := gridCounters[j]
    gridCounters[j]++
    gridCells[j][p] := k
```

Notes:

- `gridCells` and `gridCounters` are in global memory.
- Global memory access is random and coalesced access will not work.

Creating grid with atomic operations

Physical Simulations

Simon Green. CUDA particles. www.nvidia.com/cuda, 2008

```
forall k in parallel do
    j := calcCellNo(k)
    p := gridCounters[j]
    gridCounters[j]++
    gridCells[j][p] := k
```

Notes:

- `gridCells` and `gridCounters` are in global memory.
- Global memory access is random and coalesced access will not work.
- Updating arrays may be done by many threads in the same time – `atomicAdd` must be used.

  `p = atomicAdd( &gridCounters[j], 1 )`

Creating grid with atomic operations

Physical Simulations

Simon Green. CUDA particles. www.nvidia.com/cuda, 2008

```
forall k in parallel do
    j := calcCellNo(k)
    p := gridCounters[j]
    gridCounters[j]++
    gridCells[j][p] := k
```

Notes:

- `gridCells` and `gridCounters` are in global memory.
- Global memory access is random and coalesced access will not work.
- Updating arrays may be done by many threads in the same time – `atomicAdd` must be used.

  `p = atomicAdd( &gridCounters[j], 1 )`

Creating grid with atomic operations

Physical Simulations

Simon Green. CUDA particles. www.nvidia.com/cuda, 2008

```
forall k in parallel do
    j := calcCellNo(k)
    p := gridCounters[j]
    gridCounters[j]++
    gridCells[j][p] := k
```

Notes:

- `gridCells` and `gridCounters` are in global memory.
- Global memory access is random and coalesced access will not work.
- Updating arrays may be done by many threads in the same time – `atomicAdd` must be used.

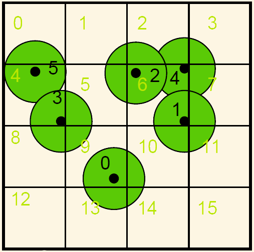
  `p = atomicAdd( &gridCounters[j], 1 )`

In some devices atomic functions must be turned on by compiling with `-arch sm_11` nvcc option.

Creating grid without atomic operations I

Physical Simulations

Simon Green. CUDA particles. www.nvidia.com/cuda, 2008



Index	Unsorted list (cell id, particle id)	List sorted by cell id	Cell start
0	(9, 0)	(4, 3)	
1	(6, 1)	(4, 5)	
2	(6, 2)	(6, 1)	
3	(4, 3)	(6, 2)	
4	(6, 4)	(6, 4)	0
5	(4, 5)	(9, 0)	
6			2
7			
8			
9			5
10			
11			
12			
13			
14			
15			

Creating grid without atomic operations II

Physical Simulations

Simon Green. CUDA particles. www.nvidia.com/cuda, 2008

```
forall k in parallel do
   j := calcCellNo(k)
   particlesGrid[k].cellNo := j
   particlesGrid[k].particle := k

parallelSortByCellNo( particlesGrid )

forall 0 < k in parallel do
   if particlesGrid[k].cellNo <> particlesGrid[k − 1].cellNo
      cellStart[particlesGrid[k].cellNo] = k
cellStart[particlesGrid[0].cellNo] = 0
```

Creating grid without atomic operations II

Simon Green. CUDA particles. www.nvidia.com/cuda, 2008

```
forall k in parallel do
   j := calcCellNo(k)
   particlesGrid[k].cellNo := j
   particlesGrid[k].particle := k

parallelSortByCellNo( particlesGrid )

forall 0 < k in parallel do
   if particlesGrid[k].cellNo <> particlesGrid[k - 1].cellNo
      cellStart[particlesGrid[k].cellNo] = k
cellStart[particlesGrid[0].cellNo] = 0
```

Notes:

- The method with sorting is about 40% faster than the previous one.

Part 3 – Algorithms

Barnes Hut force-calculation for n-body

Physical Simulations

The tree-based algorithm reduces $O(n^2)$ to $O(n \log n)$
It is a challenge since:

1. it repeatedly builds and traverses an irregular tree-based data structure,
2. it performs a lot of pointer-chasing memory operations,
3. it is typically expressed recursively.

General schema of the algorithm

Physical Simulations

1. Read input data and transfer to GPU
2. for each timestep do:
   - 2.1 Compute bounding box around all bodies
   - 2.2 Build hierarchical decomposition by inserting each body into octree
   - 2.3 Summarize body information in each internal octree node
   - 2.4 Approximately sort the bodies by spatial distance
   - 2.5 Compute forces acting on each body with help of octree
   - 2.6 Update body positions and velocities
3. Transfer result to CPU and output

Based on:

Martin Burtscher and Keshav Pingali. An efficient cuda implementation of the tree-based barnes hut n-body algorithm. *GPU Computing Gems Emerald Edition*, 12 2011

Memory structures

Physical Simulations

- n-body objects converted to SoA: fields grouped in separated arrays
- Allocate bodies at the beginning and the cells at the end of the arrays
- Use an index of -1 as a null pointer.
- Advantages:
  - A simple comparison of the array index with the number of bodies determines whether the index points to a cell or a body.
  - In some code sections, we need to find out whether an index refers to a body or to null. Because -1 is also smaller than the number of bodies, a single integer comparison suffices to test both conditions.

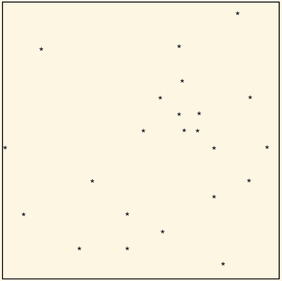
BC array:

b_1	b_2	b_3	...	...	c_3	c_2	c_1

General schema of the algorithm – kernels

Kernel 1

Compute bounding box around all bodies:

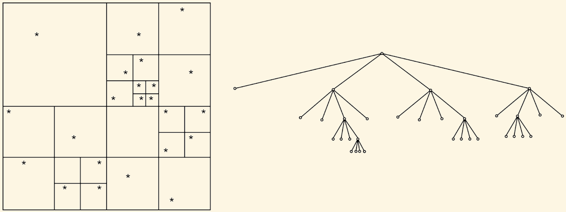


- Break data into equal chunks.
- Perform reduction operation in blocks.
- Use `min()` and `max()` since they are faster than `if...` statement.
- The last block combines results and generates the root of the tree.

General schema of the algorithm – kernels

Kernel 2

Build hierarchical decomposition by inserting each body into octree:



- Implements an iterative tree-building algorithm that uses lightweight locks
- Bodies are assigned to the blocks and threads within a block in round-robin fashion.
- Each thread inserts its bodies one after the other by:
  - traversing the tree from the root to the desired last-level cell
  - attempting to lock the appropriate child pointer (an array index) by writing an otherwise unused value to it using an atomic operation
  - If the lock succeeds, the thread inserts the new body and release the lock

General schema of the algorithm – kernels

Kernel 2 – pseudocode

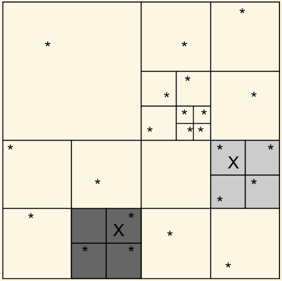
Repeat to get the success flag true:

```
// initialize
cell = find_insertion_point(body); // nothing is locked, cell cached
      for retries
child = get_insertion_index(cell, body);
if (child != locked) {
    if (child == atomicCAS(&cell[child], child, lock)) {
        if (child == null) {
            cell[child] = body; // insert body and release lock
        } else {
            new_cell =...; // atomically get the next unused cell
            // insert the existing and new body into new cell
            threadfence(); // make sure new cell subtree is visible
            cell[child] = new_cell; // insert new cell and release
                 lock
        }
       success = true; // flag indicating that insertion succeeded
    }
}
syncthreads(); // wait for other warps to finish insertion
```

General schema of the algorithm – kernels

Kernel 3

Summarize body information in each internal octree node:



- traverses the unbalanced octree from the bottom up to compute the center of gravity and the sum of the masses of each cell's children
- Cells are assigned to blocks and threads in a round-robin fashion.
- Ensure load-balance, Start from leaves so avoid deadlocks, Allow some coalescing

General schema of the algorithm – kernels

Kernel 3 – pseudocode

```
// initialize
if (missing == 0) {
    // initialize center of gravity
    for (/*iterate over existing children*/) {
        if (/*child is ready*/) {
            // add its contribution to center of gravity
        } else {
            // cache child index
            missing++;
        }
    }
}
if (missing != 0) {
    do {
        if (/*last cached child is now ready*/) {
            // remove from cache and add its contribution to center of gravity
            missing--;
        }
    } while (/*missing changed*/ && (missing != 0));
}
if (missing == 0) {
    // store center of gravity
    __threadfence(); // make sure center of gravity is visible
    // store cumulative mass
    success = true; // local flag indicating that computation for cell is done
}
```

General schema of the algorithm – kernels

Kernel 4

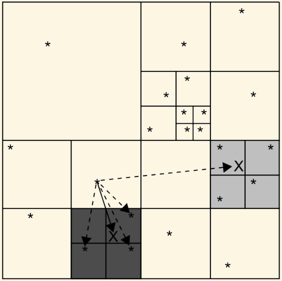
Approximately sort the bodies by spatial distance.
Kernel 4 is not needed for correctness but for optimization.

- It is done by in-order traversal of the tree.
- Typically places spatially close bodies close together.
- It is based on the number of bodies in each subtree, which was computed in kernel 3.
- It concurrently places the bodies into an array such that the bodies appear in the same order in the array as they would during an in-order traversal of the octree.

General schema of the algorithm – kernels

Kernel 5

Compute forces acting on each body with help of octree:



- For each body, the corresponding thread traverses some prefix of the octree to compute the force acting upon this body.

General schema of the algorithm – kernels

Kernel 5 – pseudocode

```
// precompute and cache info
// determine first thread in each warp
for (/*sorted body indexes assigned to me*/) {
    // cache body data
    // initialize iteration stack
    depth = 0;
    while (depth >= 0) {
        while (/*there are more nodes to visit*/) {
            if (/*I'm the first thread in the warp*/) {
                // move on to next node
                // read node data and put in shared memory
            }
        __threadfence_block();
        if (/*node is not null*/) {
            // get node data from shared memory
            // compute distance to node
            if ((/*node is a body*/) || all(/*distance >= cutoff*/)) {
                // compute interaction force contribution
            } else {
                depth++; // descend to next tree level
                if (/*I'm the first thread in the warp*/) {
                    // push node's children onto iteration stack
                }
                __threadfence_block();
            }
        } else {
            depth = max(0, depth-1); // early out because remaining nodes are also null
        }
    }
    depth--;
}
// update body data
}
```

General schema of the algorithm – kernels

Kernel 6

Update velocities and positions of all bodies:

- It is a straightforward, fully coalesced, nondivergent streaming kernel.
- As in the other kernels, the bodies are assigned to the blocks and threads within a block in round-robin fashion.

Part 3 – Algorithms

Summary of optimizations

Physical Simulations

MAIN MEMORY

Minimize Accesses

- Let one thread read common data and distribute data to other threads via shared memory
- When waiting for multiple data items to be computed, record which items are ready and only poll the missing items
- Cache data in registers or shared memory
- Use thread throttling (see control-flow section)

Summary of optimizations

Physical Simulations

MAIN MEMORY

Maximize Coalescing

- Use multiple aligned arrays, one per field, instead of arrays of structs or structs on heap
- Use a good allocation order for data items in arrays

Reduce Data Size

- Share arrays or elements that are known not to be used at the same time

Minimize CPU/GPU Data Transfer

- Keep data on GPU between kernel calls
- Pass kernel parameters through constant memory

Summary of optimizations

Physical Simulations

CONTROL FLOW

Minimize Thread Divergence

- Group similar work together in the same warp

Combine Operations

- Perform as much work as possible per traversal, i.e., fuse similar traversals

Throttle Threads

- Insert barriers to prevent threads from executing likely useless work

Minimize Control Flow

- Use compiler pragma to unroll loops

Summary of optimizations

Physical Simulations

LOCKING

Minimize Locks

- Lock as little as possible (e.g., only a child pointer instead of entire node, only last node instead of entire path to node)

Use Lightweight Locks

- Use flags (barrier/store and load) where possible
- Use atomic operation to lock but barrier/store or just store to unlock

Reuse Fields

- Use existing data field instead of separate lock field

Summary of optimizations

Physical Simulations

HARDWARE

Exploit Special Instructions

- Use min, max, threadfence, threadfence block, syncthreads, all, rsqft, etc. operations

Maximize Thread Count

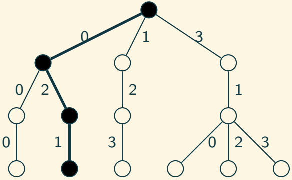
- Parallelize code across threads
- Limit shared memory and register usage to maximize thread count

Part 3 – Algorithms

Building a radix search tree

Radix search tree

At each level we consider r bits of the vectors.
We get 2^r possible children of each node.



$r = 2.$

x	$\widetilde{x}$
00 00 00	000
00 10 01	**021**
01 10 11	123
11 01 00	310
11 01 10	312
11 01 11	313

Parallel Tree Building

Top-down (level 0)

- sort input vectors

Parallel Tree Building

Top-down (level 0)

- sort input vectors
- transpose data vectors – columns are rows now

Parallel Tree Building

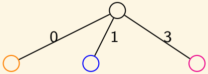
Top-down (level 0)

- sort input vectors
- transpose data vectors – columns are rows now

Parallel Tree Building

Top-down (level 0)

- sort input vectors
- transpose data vectors – columns are rows now

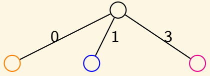


$$\widetilde{x}^T \left| \begin{array}{l} \color{orange}{00}\color{blue}{1}\color{magenta}{333} \\ 022111 \\ 013023 \end{array} \right.$$

Parallel Tree Building

Top-down (level 0)

- sort input vectors
- transpose data vectors – columns are rows now



$$\widetilde{x}^T \left| \begin{array}{l} 001333 \\ 022111 \\ 013023 \end{array} \right.$$

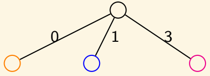
- Marking existence of children

c_0	c_1	c_2	c_3
1	1	0	1

Parallel Tree Building

Top-down (level 0)

- sort input vectors
- transpose data vectors – columns are rows now



$$\tilde{x}^T \;\Bigg|\; \begin{matrix} 001333 \\ 022111 \\ 013023 \end{matrix}$$

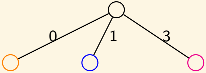
- Marking existence of children
- Pre-scan array

	c_0	c_1	c_2	c_3
Marking existence of children	1	1	0	1
Pre-scan array	0	1	2	2

Parallel Tree Building

Top-down (level 0)

- sort input vectors
- transpose data vectors – columns are rows now



$\widetilde{x}^T$

001333
022111
013023

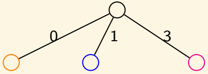
- Marking existence of children
- Pre-scan array
- Number of children in the next level: $2 + 1 = 3$

	c_0	c_1	c_2	c_3
Marking existence of children	1	1	0	1
Pre-scan array	0	1	2	2

Parallel Tree Building

Top-down (level 0)

- sort input vectors
- transpose data vectors – columns are rows now



$\widetilde{x}^T$

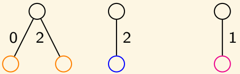
001333
022111
013023

- Marking existence of children
- Pre-scan array
- Number of children in the next level: $2 + 1 = 3$
- In parallel for each existing child node(blocks):

	c_0	c_1	c_2	c_3
Marking existence of children	1	1	0	1
Pre-scan array	0	1	2	2

Parallel Tree Building

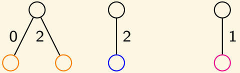
Top-down (level 1)



$$\widetilde{x}^T \left| \begin{array}{l} 001333 \\ 022111 \\ 013023 \end{array} \right.$$

Parallel Tree Building

Top-down (level 1)



$\widetilde{x}^T$

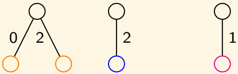
001333
022111
013023

- Marking existence of children

c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}
1	0	1	0	0	0	1	0	0	1	0	0

Parallel Tree Building

Top-down (level 1)



$\widetilde{x}^T$

001333
022111
013023

- Marking existence of children

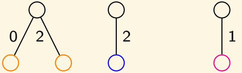
c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}
1	0	1	0	0	0	1	0	0	1	0	0

- Pre-scan array

0	1	1	2	2	2	2	3	3	3	4	4

Parallel Tree Building

Top-down (level 1)



$\widetilde{x}^T$	001333
	022111
	013023

- Marking existence of children

c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}
1	0	1	0	0	0	1	0	0	1	0	0

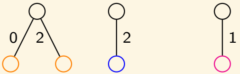
- Pre-scan array

0	1	1	2	2	2	2	3	3	3	4	4

- Number of children in the next level: $4 + 0 = 4$

Parallel Tree Building

Top-down (level 1)



$$\widetilde{x}^T \left| \begin{array}{l} 001333 \\ 022111 \\ 013023 \end{array} \right.$$

- Marking existence of children

c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}
1	0	1	0	0	0	1	0	0	1	0	0

- Pre-scan array

0	1	1	2	2	2	2	3	3	3	4	4

- Number of children in the next level: $4 + 0 = 4$
- Repeat in parallel for each existing child node (blocks)...

Bibliography

Guy E Blelloch. Prefix sums and their applications, 1990.

Martin Burtscher and Keshav Pingali. An efficient cuda implementation of the tree-based barnes hut n-body algorithm. *GPU Computing Gems Emerald Edition*, 12 2011.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

Nvidia cuda sdk. www.nvidia.com/cuda.

Daniels220. English Wikipedia, CC BY-SA 3.0. https://commons.wikimedia.org/w/index.php?curid=6678551.

Simon Green. CUDA particles. www.nvidia.com/cuda, 2008.

Materiały sponsorowane przez:

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Politechnika Warszawska

Unia Europejska
Europejski Fundusz Społeczny