Introduction to Data-Parallel Algorithms

Getting the Most out of your GPU

Imre Palik

imre.palik@morganstanley.com

Morgan Stanley

The views expressed in this presentation are those of the author and, therefore, do not necessarily reflect the views of Morgan Stanley

Data-Parallel Algorithms – Why?

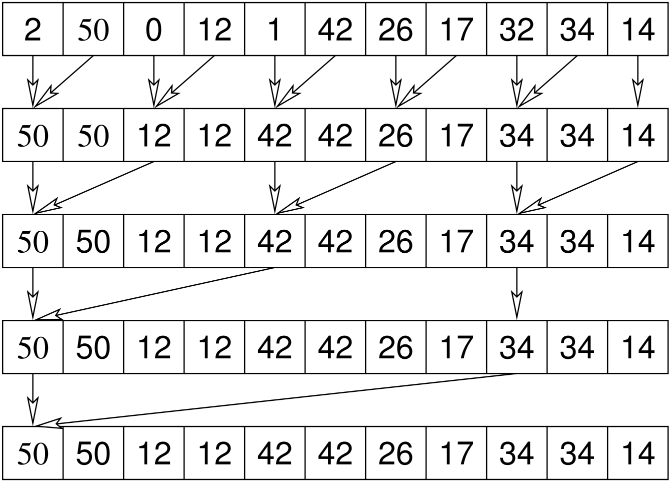
- Algorithm design techniques are handy when solving complex problems
- One can increase the parallelism of seemingly serial (sub) problems
- No good libraries for writing custom kernels

Maximum Element of an Array


```
template typename<T>
T
max(size_t len, T array[])
{
  assert(len);
  T rv = array[0];
  for (size_t c = 1; c < len; c++)
    if (array[c] > rv)
      rv = array[c];
  return rv;
}
```

Can we do this in parallel?

Maximum Element of an Array – Parallel



Morgan Stanley

Reduction

Definition

A reduction operation takes a binary associative operator $\oplus$ with identity i, and an ordered set $[a_0, a_1, \ldots, a_{n-1}]$ of n elements, and returns the value $a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$.

Parallel Reduction – The Code


```
__global__ void sum_reduce(double * work) {
  __shared__ double w_s[];
  w_s[threadIdx.x] = work[threadIdx.x];
  __syncthreads();
  for (unsigned d = 2, len = blockDim.x/2; len > 0;
       len /= 2, d *= 2)
  {
    if (threadIdx.x < len)
      w_s[d * threadIdx.x] = w_s[d * threadIdx.x]
        + w_s[d * threadIdx.x + d/2];
    __syncthreads();
  }
  if (!threadIdx.x) *work = w_s[0];
}
```

If you have More Data Than Threads . . .


```
__device__ double
sum_reduce(double * work, const unsigned len,
           const unsigned nthreads, const unsigned tid
{ // first phase
  const unsigned step = len/nthreads
    + (len%nthreads > 0);
  double acc = work[tid * step];
  for (int c = 0; c < step && tid * step + c < length;
       c++)
    acc = acc + work[tid * step + c];
  work[tid * step] = acc;
  __syncthreads();
  // second phase
}
```

Sidetrack – nVidia GPU Architecture

Hierarchical synchronisation structure.

Warp Threads running on the same vector processor at the same time. Synchronised by the hardware

Threadblock Threads running on the same vector processor. Explicit synchronisation possible.

Grid All the threads executing the same kernel. Synchronised at kernel launches.

Two-Level Reduction

Problems with implementing parallel reduction on GPUs:

- Parallel reduction needs synchronisation.
- Grid-wide synchronisation is really expensive
- Block-wide synchronisation is relatively cheap.

Two-Level Reduction

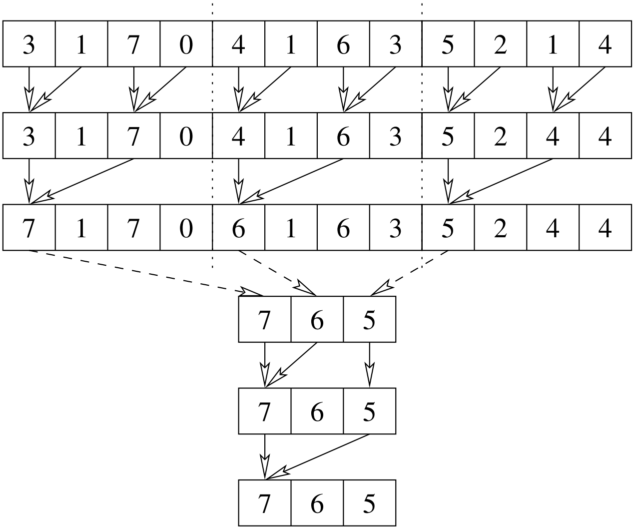
Problems with implementing parallel reduction on GPUs:

- Parallel reduction needs synchronisation.
- Grid-wide synchronisation is really expensive
- Block-wide synchronisation is relatively cheap.

Solution:

1. Parallel reduction for subarrays in each threadblock
2. Parallel reduction on the results in a single block

Two-Level Reduction – Cont.



Three-Level Reduction – Warp Level


```
__device__ double
sum_reduce_w (double *w, unsigned len) {
  const unsigned wid = threadIdx.x%warpSize;
  while (len)
  {
    if (wid < len/2)
      w[wid] = w[2 * wid] + w[2 * wid + 1];
    len /= 2;
  }
}
```

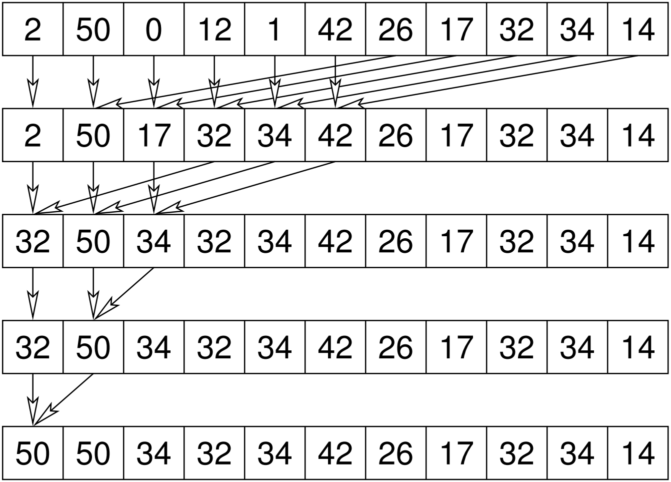
Three-Level Reduction – Warp Level Cont.


```
__device__ double
sum_reduce_w (double *w, unsigned len) {
  const unsigned wid = threadIdx.x%warpSize;
  switch (len) {
  case 64:
      w[wid] = w[2 * wid] + w[2 * wid + 1];
  case 32:
    if (wid < 16)
      w[wid] = w[2 * wid] + w[2 * wid + 1];
  case 16:
    if (wid < 8)
      w[wid] = w[2 * wid] + w[2 * wid + 1];
  case 8:
    // ...
}
```

Three-Level Reduction – Cont.


```
__device__ double
sum_reduce_b (double *w) {
  double val =
    sum_reduce_w(w + (threadIdx.x/warpSize * 32), 32);
  __syncthreads();
  if (!(threadIdx.x%warpSize))
    w[threadIdx.x/warpSize] = val;
  __syncthreads();
  if (threadIdx.x < warpSize)
    val = sum_reduce_w(w, blockDim.x/warpSize);
  return val;
}
```

Commutative Reduction



Commutative Reduction – the Code


```
__device__ double
sum_reduce_w(double * work, unsigned len) {
  const unsigned wid = threadIdx.x%warpSize;
  switch (len) {
  case 64:                  work[wid] += work[wid + 32];
  case 32: if (wid < 16) work[wid] += work[wid + 16];
  case 16: if (wid <  8) work[wid] += work[wid +  8];
  case  8: if (wid <  4) work[wid] += work[wid +  4];
  case  4: if (wid <  2) work[wid] += work[wid +  2];
  case  1: if (!wid)     work[wid] += work[wid +  1];
  }
  return work[0];
}
```

```
for (unsigned c = 1; c <= len; c++)
  out[c] = out[c - 1] + in[c - 1];
```

Prefix Sums and their Friends – Cont.


```
for (unsigned c = 1; c <= len; c++)
  out[c] = f(out[c - 1], in[c - 1]);
```

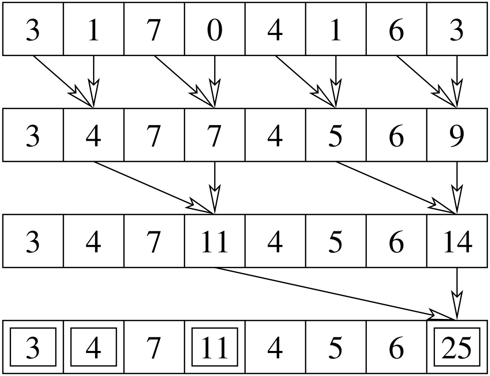
The Name of the Game

Definition

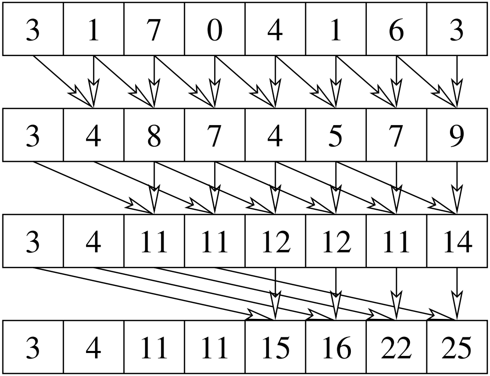
The all-prefix-sum (scan) operation takes a binary associative operator $\oplus$ and an ordered set $[a_0, a_1, \ldots, a_{n-1}]$ of n elements, and returns the value

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

Back to the Reduction



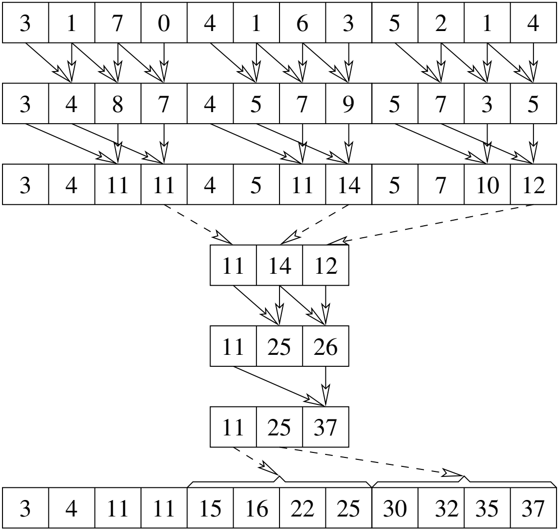
Naive Parallel Scan



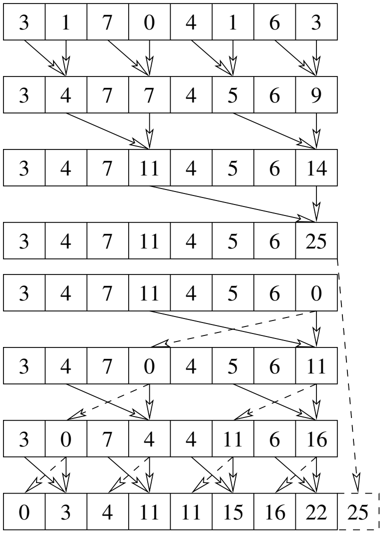
Naive Parallel Scan – The Code


```
__device__ double
sum_scan_w (double *w, unsigned len) {
  const unsigned wid = threadIdx.x%warpSize;
  for (unsigned offset = 1; offset < len; offset *= 2)
    if (wid + offset < len)
      w[wid + offset] += w[wid];
}
```

Partitioned Naive Parallel Scan



Brent-Kung Style Scan



Parallelisable Recurrences

$$x_i = \begin{cases} b_0 & i = 0 \\ (x_{i-1} \otimes a_i) \oplus b_i & 0 < i < n \end{cases}$$

1. $\oplus$ is associative
2. $\otimes$ is semi-associative (exists $\odot$ associative operator, such that $(a \otimes b) \otimes c = a \otimes (b \odot c)$)
3. $\otimes$ distributes over $\oplus$

Parallelisable Recurrences – Cont.

Theorem

The x_i recurrence defined on the previous slide can be solved by scan.

Proof.

Let $c_i = [a_i, b_i]$ and define $*$ by
$c_i * c_j = [c_{i,a} \odot c_{j,a}, (c_{i,b} \otimes c_{j,a}) \oplus c_{j,b}]$. Then $*$ is associative.
Define $s_i = [y_i, x_i]$, where $y_i = \begin{cases} a_0 & i = 0 \\ y_{i-1} \odot a_i & 0 < i < n \end{cases}$. Then

$$s_i = \begin{cases} c_0 & i = 0 \\ s_{i-1} * c_i & 0 < i < n \end{cases}$$

□

Segmented Scan

$$
\begin{array}{lclllllllll}
a & = & [5 & 1 & 3 & 4 & 3 & 9 & 2 & 6] \\
f & = & [1 & 0 & 1 & 0 & 0 & 0 & 1 & 0] \\
\hline
\text{segmented +-scan} & = & [5 & 6 & 3 & 7 & 10 & 19 & 2 & 8] \\
\text{segmented max-scan} & = & [5 & 5 & 3 & 4 & 4 & 9 & 2 & 6]
\end{array}
$$

Segmented Scan → Scan

Segmented scans satisfy the recurrence

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

where

$$x \times f = \begin{cases} I_\oplus & f = 1 \\ x & f = 0 \end{cases}$$

$\times$ is semi-associative with *logical or* as the companion operator.

Radix Sort


```
void
radix_sort(long * array, size_t len,
           unsigned n_digits){
  for (unsigned c = 0; c < n_digits; c++)
    stable_sort_on_digit(array, len, c);
}
```

Sequential Counting Sort


```
void
counting_sort(long * a, long * b, unsigned len,
              unsigned k) {
  long ls[k];
  memset(ls, 0, k * sizeof(long));
  for (unsigned c = 0; c < len; c++)
    ls[a[c]]++;
  for (unsigned c = 1; c < len; c++)
    ls[c] += ls[c - 1];
  for (signed s = len - 1; s >= 0; s--) {
    b[ls[a[s]]] = a[s];
    ls[a[s]]--;
  }
}
```

Binary Counting Sort (Split)

```
void
split(long * a, unsigned * flags, unsigned l,
      unsigned * idown, unsigned * iup,
      unsigned * idx) {
  idown = enumerate(not(flags)) - 1;
  iup = idown[l - 1] + enumerate(flags);
  index = flags? iup : idown;         // vector op
  permute(a, index);
}
```

A	= [	5	7	3	1	4	2	7	2]
Flags	= [	T	T	T	T	F	F	T	F]
I-down	= [	-1	-1	-1	-1	0	1	1	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]

Binary Radix Sort

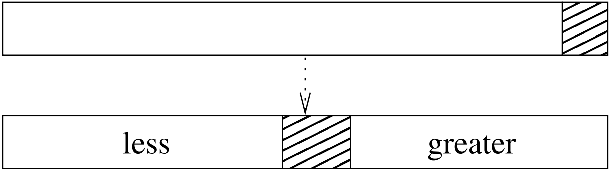
$$
\begin{aligned}
A &= [5 \quad 7 \quad 3 \quad 1 \quad 4 \quad 2 \quad 7 \quad 2] \\
A < 0 > &= [T \quad T \quad T \quad T \quad F \quad F \quad T \quad F] \\
A \leftarrow \text{split}(A, A < 0 >) &= [4 \quad 2 \quad 2 \quad 5 \quad 7 \quad 3 \quad 1 \quad 7] \\
A < 1 > &= [F \quad T \quad T \quad F \quad T \quad T \quad F \quad T] \\
A \leftarrow \text{split}(A, A < 1 >) &= [4 \quad 5 \quad 1 \quad 2 \quad 2 \quad 7 \quad 3 \quad 7] \\
A < 2 > &= [T \quad T \quad F \quad F \quad F \quad T \quad F \quad T] \\
A \leftarrow \text{split}(A, A < 1 >) &= [1 \quad 2 \quad 2 \quad 3 \quad 4 \quad 5 \quad 7 \quad 7]
\end{aligned}
$$

Quicksort


```
void
quicksort(double * b, double * e) {
  if (b < e)
  {
    size_t p = partition(b, e);
    quicksort(b, b + p);
    quicksort(b + p + 1, e);
  }
}
```

Sequential Partition

```
unsigned
partition(double * b, double * e) {
  double p = *(e - 1);
  unsigned i = 0;
  for (unsigned c = 0; c < e - b; c++)
    if (b[c] < = p)
      swap(b + i++, b + j);
  swap(b + i, e - 1);
  return i;
}
```



Parallel Quicksort


```
void
parallel_quicksort(double * a, bool * f, unsigned l){
  while(!sorted(a, l))
    parallel_partition(a, f, l, p, tf));
}

void
parallel_partition(double * k, bool * sf, unsigned l,
                   double * p, signed char * f) {
  seg_copy(p, k, sf);                    // with scan
  f = k < p? -1 : (k == p? 0 : 1); // vector compare
  seg_split(k, f, sf);                   // 3-way split
  sf |= new_seg_flags(k, p);
}
```

Parallel Quicksort – Example

Key =	6.4	9.2	3.4	1.6	8.7	4.1	9.2	3.4
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 $\leftarrow$ split(Key, F) =	3.4	1.6	4.1	3.4	6.4	9.2	8.7	9.2
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 $\leftarrow$ split(Key, F) =	1.6	3.4	3.4	4.1	6.4	8.7	9.1	9.2
Flags =	1	1	0	1	1	1	1	0