Optimising Purely Functional GPU Programs

Trevor L. McDonell
University of New South Wales

Jointly with
Manuel M. T. Chakravarty
Gabriele Keller
Ben Lippmeier
High Level Languages


High Level Languages

Composite data structures

Immutable structures

Boxed values

Higher-order functions & closures

Polymorphism & generics
High Level Languages

- GPUs
- multicore CPU
- Cluster
High Level Languages

Function pointers
Control flow

Memory access patterns
Decomposition

Data distribution

GPUs  
multicore CPU  
Cluster
High Level Languages

- Function pointers
- Control flow
- Memory access patterns
- Decomposition
- Data distribution

Efficient code?
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q ls =  q : (case ls of
                   []    -> []
                   x:xs  -> scanl f (f q x) xs)
Fast Scan Algorithms on Graphics Processors

Yuri Dotsenko  Naga K. Govindaraju  Peter-Pike Sloan  Charles Boyd  John Manferdelli

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA
{yurido, nagag, ppsloan, chasb, jmanfer}@microsoft.com

ABSTRACT

Scan, segmented scan, and parallel primitives are important data structures which have a wide range of applications. We present fast scan algorithms on Graphics Processing Units (GPUs). Our algorithms exploit shared memory to improve the performance of scan primitives for computing arbitrary segment lengths. We also present optimizations to improve the performance of segmented scans based on the context of modern GPU architectures such as memory bank conflicts. Our results indicate up to 60% higher performance than prior optimized algorithms. On segmented scans, we observed up to an order of magnitude higher performance over optimized sequential algorithms. Our algorithms may not map directly to GPUs due to complexity, e.g., pipelining, register files, several data parallel algorithms, etc. These algorithms map well to GPUs, they are not work well on general data sets. 

Keywords  Algorithms, performance.

General Terms  D.1.3: Parallel programming.
Fast Scan Algorithms on Graphics Processors

Yuri Dotsenko  Naga K. Govindaraju  Peter-Pike Sloan  Charles Boyd  John Manferdelli
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA
{yurido, nagag, ppsloan, chasb, jmanfer}@microsoft.com

ABSTRACT

Scan and segmented scan are important data

optimization primitives. They are often used in

algorithms such as quicksort and are formalized

as merge, prefix, and scan. Scan algorithms for

these primitives on graphics processing units

(GPUs) are of practical interest for rendering

core algorithms. Our results indicate up to

10x higher performance.

We improve the performance of segmented scans based on the

algorithm, 

that the prior algorithms

in Section 2. We present fast scan algorithms that map

better to GPUs and

we give an overview of scan algorithms and the issues in mapping

them to an order of magnitude higher performance over optimized

prior state

The rest of the paper is organized as

Subject Descriptors

D.1.3 [Databases]: Parallel programming

Keywords

D.1.3 [Databases]: Parallel programming

We implemented

on input sequences with

bank conflicts and reducing the overheads in prior shared

memory performance. We further improve the

performance of segmented scans based on the

prior optimized algorithms. On s

unsegmented scans indicate up to 60% higher performance than

the Fermi architecture

as radix sort

A. S. Graham and E. Megiddo, [15] presented the first GPU

based algorithms. Our results on

the high parallelism on GPUs. Our algorithm involves low

issues

to prior state

on several GPUs

based segmented scan algorithms.

}
Fast Scan Algorithms on Graphics Processors

Yuri Dotsenko  Naga K. Govindaraju  Peter-Pike Sloan  Charles Boyd  John Manferdelli

Optimising Parallel Prefix operations for the Fermi architecture

Mark Harris  Michael Garland
NVIDIA Corporation
Recently, many scan algorithms have been designed to map to GPUs. The basic building blocks in many parallel algorithms are scan primitives such as prefix, segmented prefix, and segmented sum. These algorithms exploit the high memory bandwidth and high parallelism of modern GPU architectures. The Fermi architecture, including NVIDIA GeForce 8800 GPU, provides extensive overview of scans as building blocks of parallel algorithms. Our main contribution is a novel data representation in shared memory that maps better to the GPU memory hierarchy. We present fast scan algorithms that map better to GPUs and improve the performance of scan primitives for computing arbitrary segment lengths. We also present optimizations to improve the memory performance. We further improve the implementation of parallel primitives and formalize scan for the P-RAM model. Hensley et al. implemented an efficient parallel algorithm for the Fermi architecture, but modern GPU architectures such as memory bank conflicts. These algorithms may not map directly to GPUs due to complexity in mapping them to GPUs. We present our scan algorithms and provide optimizations to improve the performance of scan algorithms. In this paper, we analyze the parallel algorithm for the Fermi architecture. We implement our algorithms on a PC with one NVIDIA GeForce 8800 GPU and compared our results with prior optimized algorithms. On several GPUs, our results indicate up to 20% improvement over prior algorithms. We summarize the paper and present future work in Section 6.
YOU WANT A PIECE OF ME?
How about embedded languages with specialised code generation?
Accelerate
An embedded language for GPU programming
Accelerate
An embedded language for GPU programming
Accelerate
An embedded language for GPU programming
dotp xs ys =
Embedded language arrays

dotp xs ys =
Embedded language arrays

dotp xs ys = zipWith (*) xs ys
dotp \, xs \, ys = \text{fold} \, (+) \, \theta \, ( \, \text{zipWith} \, (*) \, xs \, ys \, )
Embedded language arrays

From Accelerate library

dotp \( xs \) \( ys \) = fold (\( + \)) \( \theta \) (zipWith (\( * \)) \( xs \) \( ys \))
```c
#include <accelerate_cuda.h>
typedef DIM1 DimOut;
extern "C" __global__ void zipWith
(
    const DIM1 shIn0,
    const Int64* __restrict__ arrIn0_a0,
    const DIM1 shIn1,
    const Int64* __restrict__ arrIn1_a0,
    const DIM1 shOut,
    Int64* __restrict__ arrOut_a0
)
{
    const int shapeSize = size(shOut);
    const int gridSize = blockDim.x * gridDim.x;
    int ix;

    for (ix = blockDim.x * blockIdx.x + threadIdx.x; ix < shapeSize; ix += gridSize) {
        const DimOut sh = fromIndex(shOut, ix);
        const Int64 v0 = toIndex(shIn0, shape(sh));
        const Int64 v1 = toIndex(shIn1, shape(sh));

        arrOut_a0[ix] = arrIn0_a0[v0] * arrIn1_a0[v1];
    }
}
```

From Accelerate library

```
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
```

Embedded language arrays
#include <accelerate_cuda.h>

typedef DIM1 DimOut;

extern "C" __global__ void zipWith(const DIM1 shIn0, const Int64* __restrict__ arrIn0_a0, const DIM1 shIn1, const Int64* __restrict__ arrIn1_a0, const DIM1 shOut, Int64* __restrict__ arrOut_a0) {

    const int shapeSize = size(shOut);
    const int gridSize = blockDim.x * gridDim.x;
    int ix;

    for (ix = blockDim.x * blockIdx.x + threadIdx.x; ix < shapeSize; ix += gridSize) {
        const DimOut sh = fromIndex(shOut, ix);
        const int v0 = toIndex(shIn0, shape(sh));
        const int v1 = toIndex(shIn1, shape(sh));
        arrOut_a0[ix] = arrIn0_a0[v0] * arrIn1_a0[v1];
    }
}

sdata0[threadIdx.x] = y0;
_syncthreads();

if (threadIdx.x < 32) {
    if (threadIdx.x + 32 < ix) {
        x0 = sdata0[threadIdx.x + 32];
        y0 = y0 + x0;
        sdata0[threadIdx.x] = y0;
    } 
    if (threadIdx.x + 16 < ix) {
        x0 = sdata0[threadIdx.x + 16];
        y0 = y0 + x0;
        sdata0[threadIdx.x] = y0;
    } 
    if (threadIdx.x + 8 < ix) {
        x0 = sdata0[threadIdx.x + 8];
        y0 = y0 + x0;
        sdata0[threadIdx.x] = y0;
    } 
    if (threadIdx.x + 4 < ix) {
        x0 = sdata0[threadIdx.x + 4];
        y0 = y0 + x0;
        sdata0[threadIdx.x] = y0;
    } 
    x0 = sdata0[threadIdx.x];
    y0 = y0 + x0;
    sdata0[threadIdx.x] = y0;
    __syncthreads();
}

if (threadIdx.x + 512 < ix) {
    x0 = sdata0[threadIdx.x + 512];
    y0 = y0 + x0;
    sdata0[threadIdx.x] = y0;
    __syncthreads();
}

if (threadIdx.x + 256 < ix) {
    x0 = sdata0[threadIdx.x + 256];
    y0 = y0 + x0;
    sdata0[threadIdx.x] = y0;
    __syncthreads();
}

if (threadIdx.x + 128 < ix) {
    x0 = sdata0[threadIdx.x + 128];
    y0 = y0 + x0;
    sdata0[threadIdx.x] = y0;
    __syncthreads();
}

if (threadIdx.x + 64 < ix) {
    x0 = sdata0[threadIdx.x + 64];
    y0 = y0 + x0;
    sdata0[threadIdx.x] = y0;
    __syncthreads();
}

if (threadIdx.x < 32) {
    if (threadIdx.x + 32 < ix) {
        x0 = sdata0[threadIdx.x + 32];
        y0 = y0 + x0;
        sdata0[threadIdx.x] = y0;
    } 
    if (threadIdx.x + 16 < ix) {
        x0 = sdata0[threadIdx.x + 16];
        y0 = y0 + x0;
        sdata0[threadIdx.x] = y0;
    } 
    if (threadIdx.x + 8 < ix) {
        x0 = sdata0[threadIdx.x + 8];
        y0 = y0 + x0;
        sdata0[threadIdx.x] = y0;
    } 
    if (threadIdx.x + 4 < ix) {
        x0 = sdata0[threadIdx.x + 4];
        y0 = y0 + x0;
        sdata0[threadIdx.x] = y0;
    } 
    x0 = sdata0[threadIdx.x];
    y0 = y0 + x0;
    sdata0[threadIdx.x] = y0;
    __syncthreads();
}

from(shOut);
lockDim.x * gridDim.x;

if (threadIdx.x + blockDim.x < shapeSize) {
    for (int index(shOut, ix); ix < shapeSize; ix += gridDim.x) {
        arrOut_a0[v0] * arrIn1_a0[v1];
    }
}

dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
Problem #1: Fusion
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
dotp \( \mathbf{x}s \ \mathbf{y}s \) = \text{fold} (+) \( \mathbf{0} \) \( ( \text{zipWith} (*) \ \mathbf{x}s \ \mathbf{y}s ) \)
Combined skeleton

dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
Stream fusion?

Data.Vector

( D. Coutts et. al, ICFP '07 )
Stream fusion? (D. Coutts et. al, ICFP '07)
Retain the skeleton-based structure of the program
map
zipWith
backpermute
generate
map

zipWith

backpermute

generate

Producers
map
zipWith
backpermute
generate

Internal representation: fusion friendly
map
zipWith
backpermute
generate
fold
scanl
permute

Producers

Internal representation: fusion friendly

Consumers

Embed producer into consumer skeleton
map
zipWith
backpermute
generate

diagram of Producers and Consumers:

Internal representation: fusion friendly

Embed producer into consumer skeleton
Fusing networks of skeletons
Fusing networks of skeletons

Phase 1: producer/producer fusion
Fusing networks of skeletons

Phase 2: consumer/producer fusion
Fusing networks of skeletons

Phase 2: consumer/producer fusion

Single fused skeleton
Fusing networks of skeletons

Phase 2: consumer/producer fusion

(see paper for details)
let inc = (+) 1
let inc = (+) 1
let inc = (+) 1
in
  let three = inc 2
let inc = (+) 1
in let nine = let three = inc 2
in
(*) three three
let inc = (+) 1
in let nine = let three = inc 2
   in (* three three
      in (inc nine)
let inc = (+) 1
in let nine = let three = inc 2
    in (*) three three
    in
    (-) (inc nine) nine
Problem #2: Sharing
let inc = (+) 1
in let nine = let three = inc 2
    in
    (*) three three
    in
    (-) (inc nine) nine
Without sharing this is evaluated 4 times

\[
\begin{align*}
\text{let } \text{inc} &= (\times) 1 \\
\text{in } \text{let } \text{nine} &= \text{let } \text{three} = \text{inc} \ 2 \\
\text{in} \\
\text{(*) three three} \\
\text{in} \\
\text{(-) (inc nine) nine}
\end{align*}
\]
Syntactic?
Syntactic?

(E. Axelsson, ICFP '12)
Type preserving sharing recovery

Preserve the tree structure of the program
let inc = (+) 1 
in let nine = let three = inc 2 
in 
  (*) three three 
in 
  (-) (inc nine) nine
Sharing Recovery
Sharing Recovery

Phase 1: prune shared subtrees
Sharing Recovery

Phase 1: prune shared subtrees
Sharing Recovery

Phase 1: prune shared subtrees

\[
\begin{array}{c}
\text{@1} \\
\text{@2} \\
\text{@3} \\
\text{@4} \\
\text{@5} \\
\text{@6} \\
\text{@7} \\
\text{@8} \\
\text{@9} \\
\text{@10} \\
\text{@11} \\
\text{@12}
\end{array}
\]
Sharing Recovery

Phase 2: float shared terms
Sharing Recovery

Phase 2: float shared terms
Sharing Recovery

Phase 2: float shared terms
Sharing Recovery

Phase 3: introduce binders
Sharing Recovery

**Phase 3: introduce binders**

\[
\begin{align*}
\text{let } x & = @ \\
(+) & 1 \\
\text{let } y & = \text{let } z = @ \\
@ & \text{ } z \\
(@) & x \quad 2 \\
\end{align*}
\]
Sharing Recovery

\[
T \rightarrow C
\]

\[
\begin{align*}
| x & \quad T' \quad \vdash x' \quad \vdash T'' \\
\lambda x. \; T & \quad \vdash \lambda x. \; T' \quad \vdash T'' \\
T_1 \; \& \; T_2 & \quad \vdash T_1 \; \& \; T_2' \quad \vdash T_2''
\end{align*}
\]

\[
\text{prune} :: \text{Level} \rightarrow (\text{Name} \rightarrow \text{Int}) \rightarrow T^r \rightarrow ((\text{Name} \rightarrow \text{Int}), \circ T^r)
\]

\[
\begin{align*}
\text{prune } & \ell \; \Omega \; e^r \mid \nu \in \Omega \quad = (\nu \triangleright \Omega, \; \nu) \\
\text{prune } & \ell \; \Omega \; e^r \mid \text{otherwise} \quad = \text{let } (\nu \triangleright \Omega) \; e^r \\
\text{where}
\end{align*}
\]

\[
\begin{align*}
\text{enter } & \Omega \; c \quad = (\Omega, \; c) \\
\text{enter } & \Omega \; (\lambda x. \; e) \quad = \text{let } (\Omega', \; \ell, \; e') \quad \text{prune } (\ell + 1) \; \Omega \; ([\ell/x]e) \\
\text{in } & (\Omega', \; \lambda x. \; e') \\
\text{enter } & \Omega \; (e_1 \; \& \; e_2) \quad = \text{let } (\Omega_1, \; \ell_1, \; e'_1) \quad \text{prune } \ell_1 \; \Omega \; e_1 \\
\text{in } & (\Omega_2, \; e'_1 \; \& \; e'_2)
\end{align*}
\]

\[
\Gamma \rightarrow \nu \; :: i T^r \mid \nu \; :: \nu \; :: \nu \; : \ell
\]

\[
\begin{align*}
\Gamma^r \rightarrow \nu \; :: \Gamma^r \rightarrow i T^r \\
\Gamma^r \; \text{ where}
\end{align*}
\]

\[
\begin{align*}
\nu^r & \quad ::= \nu^r \\
\Gamma^r & \quad ::= \Gamma^r \\
\lambda \nu^r. \; T^r & \quad ::= T^r \rightarrow T^r \\
\Gamma^r_1 \; \& \; T^r_2 & \quad ::= \Gamma^r_1 \rightarrow T^r_2 \\
\nu^r_1 \; :: i T^r_1 & \quad ::= i T^r_1 \\
\end{align*}
\]

\[
\text{binders } :: \text{env} \rightarrow \text{var} \rightarrow \Gamma^r \rightarrow \text{env}\Gamma^r
\]

\[
\text{binders } \text{lyt } \nu \triangleright \Omega \; e \quad = \text{let map } \text{binders } \text{lyt } \nu \triangleright \Omega \; e \quad \text{in } \text{body } \text{lyt}^{+n} (\nu, \; \nu_\ell) \; e_b
\]

\[
\text{where } n = \text{length } (\nu \triangleright \Omega)
\]

\[
\begin{align*}
\text{descend } & \quad :: \circ T^r \rightarrow (\Gamma, \; \nu^r) \\
\text{descend c} & \quad = (\ast, \; e) \\
\text{descend } & \quad (\lambda \nu. \; e) \quad = \text{let } (\Gamma, \; e') \quad \text{float } \Omega \; e \\
\text{in } & \quad \text{if } \exists \nu^r. \; (\nu^r \; :: \; \ell) \quad \in \Gamma \\
\text{then } & \quad (\Gamma \setminus \nu^r, \; \lambda \nu^r. \; e')
\end{align*}
\]

\[
\begin{align*}
\text{body } & \quad :: \text{env} \rightarrow \text{var} \rightarrow \Gamma^r \rightarrow \text{env}\Gamma^r \\
\text{body } & \quad \text{lyt } \nu \triangleright \Omega \; e \quad | \; \nu = \nu_{i, j} \quad = \text{lyt } i \\
\text{body } & \quad \text{lyt } \nu \triangleright \Omega \quad = c \\
\text{body } & \quad \text{lyt } \nu_{i, j} \; (\lambda \nu. \; e) \quad = \lambda (\text{binders } \text{lyt}^{+} (\nu, \; \nu_\ell)) \; e \\
\text{body } & \quad \text{lyt } \nu_{i, j} \; (e_1 \; \& \; e_2) \quad = (\text{binders } \text{lyt } \nu_{i, j} \; e_1) \; \& \; (\text{binders } \text{lyt } \nu_{i, j} \; e_2)
\end{align*}
\]

\[
\text{descend } (e_1 \; \& \; e_2) \quad = \text{let } (\Gamma_1, \; e'_1) \quad = \text{float } \Omega \; e_1 \\
\text{in } & \quad (\Gamma_1 \setminus e'_1, \; e'_1 \; \& \; e'_2)
\]
How fast are we going?
Dot Product

![Graph showing the run time (ms) vs. elements (millions) for different methods: Data.Vector, Accelerate -fusion, ... +fusion, Hand optimised GPU. Each method is represented by a different marker: circle, cross, triangle, and line, respectively.]
Dot Product

Run Time (ms)

Elements (millions)

Data.Vector
Accelerate -fusion
... +fusion
Hand optimised GPU

2x
Dot Product

- Data.Vector
- Accelerate -fusion
- ... +fusion
- Hand optimised GPU

Run Time (ms)

Elements (millions)
Black-Scholes options pricing

Run Time (ms) vs Options (millions)

- Accelerate -sharing
- ... +sharing
- Hand optimised GPU
Black-Scholes options pricing

Accelerate -sharing
... +sharing
Hand optimised GPU

Run Time (ms)

Options (millions)
Black-Scholes options pricing

Run Time (ms)

Accelerate -sharing
... +sharing
Hand optimised GPU

17x
0.9x
N-Body

Run Time (ms)

Bodies

Accelerate -fusion -sharing

... -fusion +sharing

... +fusion +sharing

Hand optimised GPU
N-Body

Run Time (ms) vs Bodies

Accelerate -fusion -sharing
... -fusion +sharing
... +fusion +sharing
Hand optimised GPU

11x
Summary

Embedded languages are restricted languages

Skeletons encapsulate efficient code idioms

Fusion and Sharing reduce the abstraction penalty

https://github.com/AccelerateHS/