Alchemy: Three Ways of Transmutating Programs into Circuits

Satnam Singh
Microsoft Research, Cambridge, UK
A Heterogeneous Future
universal language?

grand unification theory

polygots
Sequence analysis

Striped Smith–Waterman speeds database searches six times over other SIMD implementations

Michael Farrar

Received on June 22, 2006; revised on November 13, 2006; accepted on November 14, 2006
Advance Access publication November 16, 2006
Associate Editor: Nikolaus Rajewsky

ABSTRACT

Motivation: The only algorithm guaranteed to find the optimal local alignment is the Smith–Waterman. It is also one of the slowest due to the number of computations required for the search. To speed up the algorithm, Single-Instruction Multiple-Data (SIMD) instructions have been used to parallelize the algorithm at the instruction level.

Results: A faster implementation of the Smith–Waterman algorithm is presented. This algorithm achieved 2–8 times performance improvement over other SIMD based Smith–Waterman implementations. On a 2.0 GHz Xeon Core 2 Duo processor, speeds of >3.0 billion cell updates/s were achieved.

Availability: http://farrar.michael.googlepages.com/Smith-waterman
Contact: farrar.michael@gmail.com

search. A disadvantage introduced by processing the values vertically is that conditional branches are placed in the inner loop to compute $F$. With conditional code the execution time is dependent on the length of the query string and the database, the scoring matrix and gap penalties. A speedup of over six times was reported over an optimized non-SIMD implementation.

This paper presents a new Smith–Waterman implementation where the SIMD registers are parallel to the query sequence, but are accessed in a striped pattern. Like the Rognes implementation, the query profile is calculated once for the database search, but the conditional $F$ calculations are moved outside the inner loop. Calculations speeds of >3.0 GCUPS are achieved. This is a speedup of 2–8 times over the Wozniak and Rognes SIMD implementations.
CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment
Svetlin A Manavski*1,2 and Giorgio Valle1

Address: *CRIBI, University of Padova, Padova, Italy and 1Elsalud, Srl, Padova, Italy
Email: Svetlin A Manavski* - svetlin.manavski@cribi.unipd.it; Giorgio Valle - giorgio.valle@unipd.it
* Corresponding author

from Italian Society of Bioinformatics (BITS): Annual Meeting 2007
Naples, Italy, 26-28 April 2007

Published: 26 March 2008

This article is available from: http://www.biomedcentral.com/1471-2105/9/S2/S10
© 2008 Manavski and Valle; licensee BioMed Central Ltd.
This is an open access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Abstract

Background: Searching for similarities in protein and DNA databases has become a routine procedure in Molecular Biology. The Smith-Waterman algorithm has been available for more than 25 years. It is based on a dynamic programming approach that explores all the possible alignments between two sequences; as a result it returns the optimal local alignment. Unfortunately, the computational cost is very high, requiring a number of operations proportional to the product of the length of two sequences. Furthermore, the exponential growth of protein and DNA databases makes the Smith-Waterman algorithm unrealistic for searching similarities in large sets of sequences. For these reasons heuristic approaches such as those implemented in FASTA and BLAST tend to be preferred, allowing faster execution times at the cost of reduced sensitivity. The main motivation of our work is to exploit the huge computational power of commonly available graphic cards, to develop high performance solutions for sequence alignment.

Results: In this paper we present what we believe is the fastest solution of the exact Smith-Waterman algorithm running on commodity hardware. It is implemented in the recently released CUDA programming environment by NVIDIA. CUDA allows direct access to the hardware primitives of the last-generation Graphics Processing Units (GPU). G80. Speeds of more than 3.5 GCUPS (Giga Cell Updates Per Second) are achieved on a workstation running two GeForce 8800 GTX. Exhaustive tests have been done to compare our implementation to SSEARCH and BLAST, running on a 3 GHz Intel Pentium IV processor. Our solution was also compared to a recently published GPU implementation and to a Single Instruction Multiple Data (SIMD) solution. These tests show that our implementation performs from 2 to 30 times faster than any other previous attempt available on commodity hardware.
Methodology article

160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)
Isaac TS Li¹, Warren Shum² and Kevin Truong*¹,²

Address: ¹Institute of Biomaterials and Biomedical Engineering, University of Toronto, 164 College Street, Toronto, Ontario, M5S 3G9, Canada and ²Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, 10 King’s College Circle, Toronto, Ontario, M5S 3G4, Canada

Email: Isaac TS Li - isaac.li@utoronto.ca; Warren Shum - warren.shum@utoronto.ca; Kevin Truong* - kevin.truong@utoronto.ca

* Corresponding author

Published: 7 June 2007
Received: 19 February 2007
Accepted: 7 June 2007

This article is available from: http://www.biomedcentral.com/1471-2105/8/185

© 2007 Li et al; licensee BioMed Central Ltd.
This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Abstract

Background: To infer homology and subsequently gene function, the Smith-Waterman (SW) algorithm is used to find the optimal local alignment between two sequences. When searching sequence databases that may contain hundreds of millions of sequences, this algorithm becomes computationally expensive.

Results: In this paper, we focused on accelerating the Smith-Waterman algorithm by using FPGA-based hardware that implemented a module for computing the score of a single cell of the SW matrix. Then using a grid of this module, the entire SW matrix was computed at the speed of field propagation through the FPGA circuit. These modifications dramatically accelerated the algorithm’s computation time by up to 160 folds compared to a pure software implementation running on the same FPGA with an Altera Nios II softprocessor.

Conclusion: This design of FPGA accelerated hardware offers a new promising direction to seeking computation improvement of genomic database searching.
multiple independent multi-ported memories

fine-grain parallelism and pipelining

hard and soft embedded processors
FPGAs as Co-Processors

XD2000i FPGA in-socket accelerator for Intel FSB

XD2000F FPGA in-socket accelerator for AMD socket F

XD1000 FPGA co-processor module for socket 940
opportunity

scientific computing
data mining
search
image processing
financial analytics

challenge

Verilog
ray of light

CUDA
OpenCL

C-to-gates synthesis

ROCC
SPARK

Esterel
SHIM

Streams-C

Bluespec

ImpulseC

Accelerator
RapidMind /Ct

region types and shape analysis

formal (static) analysis
Self Imposed Constraints
Our High Level Synthesis Projects

- Kiwi: concurrent C# programs for control-oriented applications [Univ. Cambridge]
- Shape analysis: synthesis of dynamic data structures (C) [MPI and CMU]
- Accelerator/FPGA: synthesis of data parallel programs in C++ [MSR Redmond]
1. Kiwi: Multi-threaded Models in C#
2. Synthesis of C Programs using the Heap for Dynamic Data Structures

```c
while (nodePtr != null) {
    ProcessNode(nodePtr);
    nodePtr = nodePtr->next;
}
```

use shape analysis tool to prove program invariant Is(k, nodePtr, null) in separation logic or prove Is(k, nodePtr, null) && k < 4 i.e. we use at most 4 cells in the circuit

```c
for (int i=0; i<4; i++)
    ProcessNode(a[i]);
```
3. Synthesis of a data-parallel DSL in C++
1. Kiwi

- gate-level VHDL/Verilog
- structural
- parallel imperative
- imperative (C)
The Accidental Semi-colon
Concurrent Circuit Models

- rendezvous
- join patterns
- transactional memory
- data parallelism
- domain specific languages
- user applications

systems level concurrency constructs
threads, events, monitors, condition variables
Kiwi

- hardware concurrency models
- event-based simulation
- Kahn networks
- multi-clock
- synchronous data-flow

- software concurrency models
- asynchronous threads
- monitors
- events
- message passing
- priorities
Kiwi Library
Kiwi.cs

Circuit model
JPEG.cs

Visual Studio
multi-thread simulation
debugging
verification

Kiwi Synthesis

circuit implementation
JPEG.v
parallel program

C# —

Thread 1 —

Thread 2 —

Thread 3 —

C to gates —

C to gates —

C to gates —

Verilog for system
```csharp
public static int max2(int a, int b) {
    int result;
    if (a > b)
        result = a;
    else
        result = b;
    return result;
}
```

```csharp
.method public hidebysig static int32 max2(int32 a, int32 b) cil managed
{
    // Code size 12 (0xc)
    .maxstack 2
    .locals init ([0] int32 result)
    IL_0000:  ldarg.0
    IL_0001:  ldarg.1
    IL_0002:  ble.s IL_0008
    IL_0004:  ldarg.0
    IL_0005:  stloc.0
    IL_0006:  br.s IL_000a
    IL_0008:  ldarg.1
    IL_0009:  stloc.0
    IL_000a:  ldloc.0
    IL_000b:  ret
}
```

```
max2(3, 7)
```

```
stack
  7
  7
```

```
local memory
  7
  0
```
public class Channel<T>
{
    T datum;
    bool empty = true;
    public void Write(T v)
    {
        lock (this)
        {
            while (!empty)
                Monitor.Wait(this);
            datum = v;
            empty = false;
            Monitor.PulseAll(this);
        }
    }
}
public T Read()
{
    T r;
    lock (this)
    {
        while (empty)
        {
            Monitor.Wait(this);
            empty = true;
            r = datum;
            Monitor.PulseAll(this);
        }
        return r;
    }
}
Producer thread

produces output
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

one-place channel

Consumer thread

consumes input, multiplies it by 2, output it, forever

one-place channel

0, 2, 6, 8, 10, 12, 14, 16, 18
```csharp
public static void Consumer()
{
    while (true)
    {
        int i = chan1.Read();
        chan2.Write(2 * i);
        Kiwi.Pause();
    }
}

public static void Producer()
{
    for (int i = 0; i < 10; i++)
    {
        chan1.Write(i);
        Kiwi.Pause();
    }
}
```
public static void Behaviour()
{
    Thread ProducerThread = new Thread(new ThreadStart(Producer));
    ProducerThread.Start();

    Thread ConsumerThread = new Thread(new ThreadStart(Consumer));
    ConsumerThread.Start();
Filter Example

\[ y_t = \sum_{k=0}^{N-1} a_k x_{t-k} \]
public static int[] SequentialFIRFunction(int[] weights, int[] input) {
    int[] window = new int[size];
    int[] result = new int[input.Length];
    // Clear to window of x values to all zero.
    for (int w = 0; w < size; w++)
        window[w] = 0;
    // For each sample...
    for (int i = 0; i < input.Length; i++)
    {
        // Shift in the new x value
        for (int j = size - 1; j > 0; j--)
            window[j] = window[j - 1];
        window[0] = input[i];
        // Compute the result value
        int sum = 0;
        for (int z = 0; z < size; z++)
            sum += weights[z] * window[z];
        result[i] = sum;
    }
    return result;
}
Transposed Filter

\[ y_t = \sum_{k=0}^{N-1} a_k x_{t-k} \]
static void Tap(int i, byte w,
    Kiwi.Channel<byte> xIn,
    Kiwi.Channel<int> yIn,
    Kiwi.Channel<int> yout)
{
    byte x;
    int y;
    while (true)
    {
        y = yIn.Read();
        x = xIn.Read();
        yout.Write(x * w + y);
    }
}
// Connect up the taps for a transposed filter
for (int i = 0; i < size; i++)
{
    int j = i; // Quiz: why do we need the local j?
    Thread tapThread = new Thread(delegate() {
        Tap(j, weights[j],
            Xchannels[j],
            Ychannels[j],
            Ychannels[j+1]);
    });
    tapThread.Start();
}
2. Synthesis of dynamic data structures in C programs

```c
while (nodePtr != null)
{
    ProcessNode(nodePtr);
    nodePtr = nodePtr->next;
}
```

use shape analysis tool to prove program invariant `ls(k, nodePtr, null)` in separation logic or prove `ls(k, nodePtr, null) && k<4` i.e. we use at most 4 cells in the circuit

```
for (int i=0; i<4; i++)
    ProcessNode(a[i]);
```
malloc
<table>
<thead>
<tr>
<th>Program</th>
<th>Bound</th>
<th>ALUTs</th>
<th>Registers</th>
<th>Block Mem</th>
<th>Blocks</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge</td>
<td>$8 \times n_1 + 8 \times n_2$</td>
<td>5,157</td>
<td>4,694</td>
<td>8,192</td>
<td>2</td>
<td>90MHz</td>
</tr>
<tr>
<td>prio</td>
<td>$8 \times n$</td>
<td>5,859</td>
<td>4,598</td>
<td>4,096</td>
<td>1</td>
<td>83MHz</td>
</tr>
<tr>
<td>packet</td>
<td>$8 \times n + 8$</td>
<td>9,413</td>
<td>9,158</td>
<td>8,192</td>
<td>2</td>
<td>76MHz</td>
</tr>
<tr>
<td>huffman</td>
<td>$48 \times n_1 - 20$</td>
<td>20,678</td>
<td>11,116</td>
<td>12,288</td>
<td>3</td>
<td>76MHz</td>
</tr>
<tr>
<td>bst_dict</td>
<td>$24 \times n_1$</td>
<td>5,786</td>
<td>5,660</td>
<td>8,192</td>
<td>2</td>
<td>125MHz</td>
</tr>
</tbody>
</table>

```c
void prio(int n, in_signal i, out_signal o) {
    LINK *tmp, *c, *buffer;
    assume(n > 0);
    while (1) {
        buffer = NULL;
        // Build up an n-sized sorted buffer
        for (int k=0; k<n; k++) {
            buffer = sorted_insert(input(i), buffer);
        }
        // Send the sorted list to the output and
        // deallocate the buffer as we walk it
        c = buffer;
        while(c != NULL) {
            output(o, c->data);
            tmp = c;
            c = c->next;
            free(tmp);
        }
    }
}
```

```c
LINK * sorted_insert(int data, LINK *l) {
    LINK * elem = l;
    LINK * prev = NULL;
    LINK * x = (LINK*)malloc(sizeof(LINK));
    assert(x != NULL);
    x->data = data;
    while (elem != NULL) {
        if (elem->data >= x->data) {
            x->next = elem;
            if (prev == NULL) { l = x; return l; }
            prev->next = x;
            return l;
        }
        prev = elem;
        elem = elem->next;
    }
    x->next = elem;
    if (prev == NULL) { l = x; return l; }
    prev->next = x;
    return l;
}
```
3. Accelerator

C++, C#, F# data parallel descriptions in a DSL

- FPGA hardware (VHDL)
- GPU code (DirectX)
- SSE3
- SIMD SMP
<table>
<thead>
<tr>
<th>Effort</th>
<th>Reward</th>
<th>Accelerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>low effort</td>
<td>low reward</td>
<td>CUDA</td>
</tr>
<tr>
<td>medium effort</td>
<td>medium reward</td>
<td>OpenCL</td>
</tr>
<tr>
<td>high effort</td>
<td>high reward</td>
<td>HLSL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DirectCompute</td>
</tr>
</tbody>
</table>
using System;
using Microsoft.ParallelArrays;

namespace AddArraysPointwise
{
    class AddArraysPointwiseDX9
    {
        static void Main(string[] args)
        {
            var x = new FloatParallelArray(new[] {1.0F, 2, 3, 4, 5});
            var y = new FloatParallelArray(new[] {6.0F, 7, 8, 9, 10});
            var dx9Target = new DX9Target();
            var z = x + y;
            foreach (var i in dx9Target.ToArray1D(z))
                Console.Write(i + " ");
            Console.WriteLine();
        }
    }
}
using System;
using Microsoft.ParallelArrays;

namespace AddArraysPointwiseMulticore
{
    class AddArraysPointwiseMulticore
    {
        static void Main(string[] args)
        {
            var x = new FloatParallelArray(new[] {1.0F, 2, 3, 4, 5});
            var y = new FloatParallelArray(new[] {6.0F, 7, 8, 9, 10});
            var multicoreTarget = new X64MulticoreTarget();
            var z = x + y;
            foreach (var i in multicoreTarget.ToArray1D (z))
                Console.Write(i + " ");
            Console.WriteLine();
        }
    }
}
using System;
using Microsoft.ParallelArrays;

class AddArraysPointwiseFPGA 
{
    static void Main(string[] args) 
    {
        var x = new FloatParallelArray(new [] {1.0F, 2, 3, 4, 5});
        var y = new FloatParallelArray(new [] {6.0F, 7, 8, 9, 10});
        var fpgaTarget = new FPGATarget();
        var z = x + y;
        fpgaTarget.ToArray1D(z);
    }
}

open System
open Microsoft.ParallelArrays

let main(args) =
    let x = new FloatParallelArray (Array.map float32 [|1; 2; 3; 4; 5|])
    let y = new FloatParallelArray (Array.map float32 [|6; 7; 8; 9; 10|])
    let z = x + y
    use dx9Target = new DX9Target()
    let zv = dx9Target.ToArray1D(z)
    printf "%A\n" zv
    0
FPA pa = new FPA(bitmap);

// Convolve in X direction
FPA rX = new FPA(0, pa.Shape);

for (int i = 0; i < kernel.Length; i++)
{
    rX += PA.Shift(pa, 0, i) * kernel[i];
}
$x \begin{array}{cccccccc}
7 & 2 & 5 & 9 & 3 & 8 & 6 & 4 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}$

$\begin{array}{cccccccc}
3 & 4 & 7 & 5 & 2 \\
4 & 3 & 2 & 1 & 0
\end{array}$

$\begin{array}{cccccccc}
6 & 20 & 63 & 15 & 16 \\
4 & 3 & 2 & 1 & 0
\end{array}$

$y = \text{sum} (a \times x) \begin{array}{cccccccc}
 & & & & & & & 120
\end{array}$
public static int[] SequentialFIRFunction(int[] weights, int[] input)
{
    int[] window = new int[size];
    int[] result = new int[input.Length];
    // Clear to window of x values to all zero.
    for (int w = 0; w < size; w++)
        window[w] = 0;
    // For each sample....
    for (int i = 0; i < input.Length; i++)
    {
        // Shift in the new x value
        for (int j = size - 1; j > 0; j--)
            window[j] = window[j - 1];
        window[0] = input[i];
        // Compute the result value
        int sum = 0;
        for (int z = 0; z < size; z++)
            sum += weights[z] * window[z];
        result[i] = sum;
    }
    return result;
}
\[ y = [y[0], y[1], y[2], y[3], y[4], y[5], y[6], y[7]] \]

shift \( (x, 0) \) = [7, 2, 5, 9, 3, 8, 6, 4] = \( x \)
shift \( (x, -1) \) = [7, 7, 2, 5, 9, 3, 8, 6]
shift \( (x, -2) \) = [7, 7, 7, 2, 5, 9, 3, 8]
\[ y = [y[0], y[1], y[2], y[3], y[4], y[5], y[6], y[7]] = a[0] \cdot [x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7]] + a[1] \cdot [x[-1], x[0], x[1], x[2], x[3], x[4], x[5], x[6]] + a[2] \cdot [x[-2], x[-1], x[0], x[1], x[2], x[3], x[4], x[5]] + a[3] \cdot [x[-3], x[-2], x[-1], x[0], x[1], x[2], x[3], x[4]] + a[4] \cdot [x[-4], x[-3], x[-2], x[-1], x[0], x[1], x[2], x[3]] \]

\[ y = a[0] \cdot \text{shift}(x, 0) + a[1] \cdot \text{shift}(x, -1) + a[2] \cdot \text{shift}(x, -2) + a[3] \cdot \text{shift}(x, -3) + a[4] \cdot \text{shift}(x, -4) \]
<table>
<thead>
<tr>
<th>shift (x, 0)</th>
<th>7</th>
<th>2</th>
<th>5</th>
<th>9</th>
<th>3</th>
<th>8</th>
<th>6</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift (x, -1)</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>shift (x, -2)</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>shift (x, -3)</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>shift (x, -4)</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

\[ \rightarrow a[0] \times \text{shift (x, 0)} \]
\[ \rightarrow a[1] \times \text{shift (x, -1)} \]
\[ \rightarrow a[2] \times \text{shift (x, -2)} \]
\[ \rightarrow a[3] \times \text{shift (x, -3)} \]
\[ \rightarrow a[4] \times \text{shift (x, -4)} \]

<table>
<thead>
<tr>
<th></th>
<th>14</th>
<th>4</th>
<th>10</th>
<th>18</th>
<th>6</th>
<th>16</th>
<th>12</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>35</td>
<td>35</td>
<td>10</td>
<td>25</td>
<td>45</td>
<td>15</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>14</td>
<td>35</td>
<td>63</td>
<td>21</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>8</td>
<td>20</td>
<td>36</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>6</td>
<td>15</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>

\[ y = \begin{bmatrix} 147 & 137 & 118 & 106 & 115 & 120 & 124 & 133 \end{bmatrix} \]
using Microsoft.ParallelArrays;
using A = Microsoft.ParallelArrays.ParallelArrays;
namespace AcceleratorSamples {
    public class Convolver {
        public static float[] Convolver1D(Target computeTarget, float[] a, float[] x) {
            var xpar = new FloatParallelArray(x);
            var n = x.Length;
            var ypar = new FloatParallelArray(0.0f, new [ ] { n });
            for (int i = 0; i < a.Length; i++)
                ypar += a[i] * A.Shift(xpar, -i);
            float[] result = computeTarget.ToArray1D(ypar);
            return result;
        }
    }
}
using Microsoft.ParallelArrays;
using A = Microsoft.ParallelArrays.ParallelArrays;
namespace AcceleratorSamples
{
    public class Convolver
    {
        public static float[,] Convolver1D_2DInput
            (Target computeTarget, float[] a, float[,] x)
        {
            var ypar = new FloatParallelArray(x);
            var n = x.GetLength(0);
            var m = x.GetLength(1);
            var ypar = new FloatParallelArray(0.0f, new[] { n, m });
            var shiftBy = new int[] { 0, 0 };
            for (var i = 0; i < a.Length; i++)
            {
                shiftBy[1] = -i;
                ypar += a[i] * A.Shift(xpar, shiftBy);
            }
            var result = computeTarget.ToArray2D(ypar);
            return result;
        }
    }
}
using System;
using Microsoft.ParallelArrays;
namespace AcceleratorSamples
{
    public class Convolver2D
    {
        static FloatParallelArray convolve(Func<int, int[]> shifts, float[] kernel, int i, FloatParallelArray a)
        {
            FloatParallelArray e = kernel[i] * ParallelArrays.Shift(a, shifts(i));
            if (i == 0)
                return e;
            else
                return e + convolve(shifts, kernel, i - 1, a);
        }

        static FloatParallelArray convolveXY(float[] kernel, FloatParallelArray input)
        {
            FloatParallelArray convolveX = convolve(i => new [] { -i, 0 }, kernel, kernel.Length - 1, input);
            return convolve(i => new [] { 0, -i }, kernel, kernel.Length - 1, convolveX);
        }

        static void Main(string[] args)
        {
            const int inputSize = 10;
            var random = new Random(42);
            var inputData = new float[inputSize, inputSize];
            for (int row = 0; row < inputSize; row++)
                for (int col = 0; col < inputSize; col++)
                    inputData[row, col] = (float)random.NextDouble() * random.Next(1, 100);
            var testKernel = new float[] { 2, 5, 7, 4, 3 };
            var dx9Target = new DX9Target();
            var inputArray = new FloatParallelArray(inputData);
            var result = dx9Target.ToArray2D(convolveXY(testKernel, inputArray));
            for (var row = 0; row < inputSize; row++)
                for (var col = 0; col < inputSize; col++)
                    Console.Write("{0} ", result[row, col]);
            Console.WriteLine();
        }
    }
}
using System;
using System.Linq;
using Microsoft.ParallelArrays;
namespace AcceleratorSamples
{
    static FloatParallelArray convolve(this FloatParallelArray a,
        Func<int, int[]> shifts,
        float[] kernel)
    {
        return kernel
            .Select((k, i) => k * ParallelArrays.Shift(a, shifts(i)))
            .Aggregate((a1, a2) => a1 + a2);
    }

    static FloatParallelArray convolveXY(this FloatParallelArray input,
        float[] kernel)
    {
        return input
cOvolve(i => new[] { -i, 0 }, kernel)
cOvolve(i => new[] { 0, -i }, kernel);
    }

    for (int col = 0; col < inputSize; col++)
        Console.Write("{0} ", result[row, col]);
    Console.WriteLine();
}
}
FPA ConvolveXY(Target &tgt, int height, int width, int filterSize, float filter[], FPA input, float *resultArray)
{
    // Convolve in X (row) direction.
    size_t dims[] = {height,width};
    FPA smoothX = FPA(0,dims, 2);
    intptr_t counts[] = {0,0};
    int filterHalf = filterSize/2;
    float scale;
    for (int i = -filterHalf; i <= filterHalf; i++)
    {
        counts[0] = i;
        scale = filter[i + filterHalf];
        smoothX += Shift(input, counts, 2) * scale;
    }

    // Convolve in Y (col) direction.
    counts[0] = 0;
    FPA result = FPA(0,dims, 2);
    for (int i = -filterHalf; i <= filterHalf; i++)
    {
        counts[1] = i;
        scale = filter[filterHalf + i];
        result += Shift(smoothX, counts, 2) * scale;
    }
    tgt.ToArray(result, resultArray, height, width, width * sizeof(float));
    return smoothX;
};
let convolveXY kernel input
= // First convolve in the X direction and then in Y
    let convolveX = convolve (fun i -> [| -i; 0 |]) kernel (kernel.Length - 1) input
    let convolveY = convolve (fun i -> [| 0; -i |]) kernel (kernel.Length - 1) convolveX
    convolveY convolveX
Convolver 2D 4000x4000 Benchmark

- Nvidia Quadro FX 580 (32 cores)
- Xeon X5550 (8 cores)
- Nvidia GeForce 8600 GTS (32 cores)
- Xeon X5550 (8 cores)
- Nvidia GeForce 8600 GTS (32 cores)
- Core2 Quad Q9550 (4 cores)
- NVIDIA Quadro NVS 160M (8 cores)
- Core2 Duo P9600 (2 cores)
- Core2 Duo P9600 (2 cores)
- Core2 Duo P9600 (2 cores)
- ATI Radeon HD 5870 (1600 cores)
- 2 x Xeon X5355 (8 cores)
- 2 x Xeon X5355 (8 cores)
- NVIDIA Quadro FX 580 (32 cores)
- Xeon X5550 (8 cores)
- Nvidia GeForce 8600 GTS (32 cores)
- Core2 Quad Q9550 (4 cores)
- NVIDIA Quadro NVS 160M (8 cores)
- Core2 Duo P9600 (2 cores)
- ATI Radeon HD 5870 (1600 cores)
- 2 x Xeon X5355 (8 cores)
Convolver 2D 4000x4000 Benchmark

- Nvidia Quadro FX 580 (32 cores)
- Xeon X5550 (8 cores)
- Nvidia GeoForce 8600 GTS (32 cores)
- Core2 Quad Q9550 (4 cores)
- NVIDIA Quadro NVS 160M (8 cores)
- Core2 Duo P9600 (2 cores)
- ATI Radeon HD 5870 (1600 cores)
- 2 x Xeon X5355 (8 cores)
- Nvidia Quadro FX 580 (32 cores)
- Xeon X5550 (8 cores)
- Nvidia GeoForce 8600 GTS (32 cores)
- Core2 Quad Q9550 (4 cores)
- NVIDIA Quadro NVS 160M (8 cores)
- Core2 Duo P9600 (2 cores)
- ATI Radeon HD 5870 (1600 cores)
- 2 x Xeon X5355 (8 cores)
x64 multicore target benchmark for 2D convolver
(24 core server Xeon E7540)
opportunity

scientific computing
data mining
search
image processing
financial analytics

Verilog

challenge
Convolver

\[ y_t = \sum_{k=0}^{N-1} a_k x_{t-k} \]
8.249ns max delay
3 x DSP48Es
63 slice registers
24 slice LUTs
BRAM
36Kbits
38,304
dual-ported

FPGA basic logic (LUTS)
DSP blocks (current max 2016)
// Compute grayscale
Target &tgt = CreateDX9Target();
float* grayF = (float*) malloc(sizeof(float) * pixels);
FPA red  = FPA(redF, rectHeight, rectWidth);
FPA green = FPA(greenF, rectHeight, rectWidth);
FPA blue  = FPA(blueF, rectHeight, rectWidth);
FPA sum = Add (77 * red, Add (151 * green, 28 * blue));
FPA gray = Divide (sum, 256);
tgt.ToArray(gray, grayF, rectHeight, rectWidth, rectWidth * sizeof(float));

// Update Photoshop image buffer
pixel = (uint8*)data;
for(int32 pixelY = 0; pixelY < rectHeight; pixelY++)
{
    for(int32 pixelX = 0; pixelX < rectWidth; pixelX++)
    {
        uint8 gray = (uint8)grayF[pixelX+pixelY*rectWidth];
        pixel[0] = (uint8)gray;
        pixel[1] = (uint8)gray;
        pixel[2] = (uint8)gray;
        pixel = pixel + 3;
        bigPixel++;
        fPixel++;
        dissolve++;
        if (maskPixel != NULL)
            maskPixel++;
    }
    pixel += (dataRowBytes - 3*rectWidth);
    bigPixel += (dataRowBytes / 2 - 3*rectWidth);
    fPixel += (dataRowBytes / 4 - 3*rectWidth);
    if (maskPixel != NULL)
        maskPixel += (maskRowBytes - rectWidth);
CUDA

//Compute and store results
__syncthreads();
#pragma unroll
for (int i = ROWS_HALO_STEPS;
     i < ROWS_HALO_STEPS + ROWS_RESULT_STEPS; i++){
    float sum = 0;

    #pragma unroll
    for(int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++)
        sum += c_Kernel[KERNEL_RADIUS - j] *
s_Data[threadIdx.y][threadIdx.x + i * ROWS_BLOCKDIM_X + j];

    d_Dst[i * ROWS_BLOCKDIM_X] = sum;
}
Satnam Singh's MSDN Blog

GPGPU and x64 Multicore Programming with Accelerator from F#

Microsoft recently released a preview of the Accelerator x86 GPU and x64 multicore programming system on Microsoft Connect. This system provides a civilized level of abstraction for writing data-parallel programs that execute on GPUs and multicore processors. An experimental FPGA target is under development.

Even on my low end graphics card I get pretty impressive performance results for the 2D convolver that is described in this blog. All 8 cores of my 64-bit Windows 7 workstation are also effectively exercised by the x64 multicore target, which exploits SIMD processor instructions and multithreading. It would be nice to say anything about performance in this blog post since I don't want to focus on how to use Accelerator from the F# functional programming language. We will work backwards by starting off with a complete implementation of a two dimensional convolver. Step by step we show how this convolver is expressed using Accelerator from F#.

First here is the beautiful implementation of a two dimensional convolver. The rest of this post explains why this code works.

```csharp
open System
open System.Linq
open Microsoft.ParallelArrays

let main() =
    let testKernel = Array [| 1; 1; 1; 1; 1; 1; 1; 1; 1 |]
    // Specify the size of each dimension of the input array
    let inputDim = 50
    // Create a pseudo-random number generator
    let random = System.Random()
    // Declare a pseudo-input data array
    let testInput = Array2D.generate inputDim inputDim (fun i j -> float2d (random.NextDouble()) * float2d (random.NextDouble()))
    // Create an accelerator float parallel array for the x input array
    let testArray = new FloatParallelArray(testInput)
    // Declare a function to convolve in the x or y direction
    let convolve shifts = let kernel = testKernel |> Array2D.toList (fun _ _ -> float2d 1.0) |> FloatParallelArray.ofArray2D
        let convolveX kernel = (fun n -> FloatParallelArray.fold kernel (fun x y -> x + y) (Array.init (Array.length shifts) (fun _ -> 0))) shifts
        let convolveY kernel = (fun n -> FloatParallelArray.fold kernel (fun x y -> x + y) (Array.init (Array.length shifts) (fun _ -> 0))) shifts
        convolveX kernel
        convolveY kernel
    // Declare a 2D convolver
    let convolve2d kernel input = FloatParallelArray.fold kernel (fun x y -> x + y) (Array.init (Array.length shifts) (fun _ -> 0))
    // First convolve in the x direction and then in the y direction
    let convolveX = convolveX (fun x y -> x + y) kernel
    let convolveY = convolveX (fun x y -> x + y) kernel
```

Tags
No tags have been created or used yet.

Archives
January 2010 (2)
December 2009 (2)
Search for “Microsoft Accelerator V2”

The MSR Accelerator v2 preview build is available for download.

What is Accelerator?

Accelerator is a high-level data parallel array-processing library which uses the parallel processing capabilities of a GPU or multi-core CPU to accelerate execution. Accelerator v1 was released to the MSR Web site in October 2006 and has been periodically updated since then. Accelerator v2 is an MSR incubation project whose goal is to validate the architecture and API approach with high quality engineering sufficient to gather real-world usage data.

What’s in Accelerator v2?

Accelerator v2 builds on Accelerator v1’s programming model and adds features that were commonly requested by Accelerator v1 users. New functionality includes:

- Accelerator v2 is written as a native-code C++ library with a managed API wrapper
- Execution on x64 multicore CPUs and DX9 GPUs
- Extensible HW target interface enabling support for execution on new devices
- Ability to execute on multiple devices within a single Accelerator instance

If you have a question or feedback about the MSR Accelerator v2 preview build, please contact us via one of the

Websites.
let rec bsort n =
  match n with
  1 -> sort2
| n -> two (bsort (n-1)) >> sndList rev >> bfly sort2 n
thread 1

thread 2

thread 3

while (nodePtr != null)
{
    ProcessNode(nodePtr);
    nodePtr = nodePtr->next;
}

use shape analysis tool to prove program invariant Is(k,nodePtr,null) in separation logic or prove Is(k,nodePtr,null) && k<=4 i.e. we use at most 4 cells in the circuit

for (int i=0; i<k; i++)
    ProcessNode(a[i]);
A Heterogeneous Future
Satnam Singh

SENIOR RESEARCHER

I currently work on two research topics:

- **Alchemy: Transmutation of Programs into Circuits.**
  - The compilation of data-parallel programs written in C++ and .NET languages like C# into FPGA circuits.
  - The Kiwi project with David Greaves which synthesizes circuits from high level circuit models written using regular multi-threaded code in languages like C#.
  - Synthesis of C programs that manipulate dynamic data structures in the heap.
  - Synthesis of data-parallel programs in C++ and C# written with using the Microsoft Accelerator V2 library into FPGAs for co-processing.
  - High level techniques for designing low level circuits. I have re-implemented my Lava system in C# and F# and I have made HLINQ which is a circuit generator for LINQ queries.
  - Evaluation and experimentation with alternative hardware description