Fast Multiplication in Binary Fields on GPUs via Register Cache

Mark Silberstein
Technion

Eli Ben-Sasson, Matan Hamilis, Eran Tromer
Brief

- Optimization methodology

  Register cache: replace shared memory by \textit{registers}

- Target applications: shared memory to cache input (e.g. stencil)
- Our case: binary field multiplication
- Result: 50\% speedup over baseline
  
x138 over a single core CPU with Intel’s CLMUL instruction
Background: execution hierarchy on NVIDIA GPUs

- GPU kernel
  - Thread block
    - Thread
  - Thread block
    - Thread
  - Thread block
    - Thread
Background: memory and execution hierarchy on NVIDIA GPUs

- Global GPU memory
- GPU kernel
- Shared memory
  - Thread block
- Registers
  - Thread
Warps: Not part of programming model
Why warp-centric programming

- MIMD divergence-free programming across warps
- SIMD-optimized lock-step execution
- “Free” synchronization among threads
Missing layer: warp cache?

Global GPU memory

GPU kernel

Shared memory

Thread block

? 

Warp

Registers

Thread
Missing layer: warp cache?

Question: Efficient data sharing among warp threads?
Shuffle: **warp-level** intrinsics
Reading other thread's registers

\[
\text{shuffle}(\text{SourceThreadID}, \text{OutputRegister})
\]

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R=0</td>
<td>R=1</td>
<td>R=2</td>
</tr>
<tr>
<td></td>
<td>out=shuffle(2, r)</td>
<td>out=shuffle(0, r)</td>
</tr>
<tr>
<td></td>
<td>out=shuffle(0, r)</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT=2</td>
<td>OUT=0</td>
<td>OUT=0</td>
</tr>
</tbody>
</table>
Shuffle vs. shared memory

- No `__syncthreads` overhead
- Significantly higher bandwidth
Shuffle vs. shared memory

- No `__syncthreads` overhead
- Significantly higher bandwidth

Challenge: programming complexity!

Application-specific algorithm modifications
This work: general *technique* to replace *input* shared memory with *shuffle*
This work: general technique to replace input shared memory with shuffle
Outline

- Code transformation example: 1-d k-stencil
- General methodology
- Binary field multiplication
- Evaluation
1-d k-stencil

\[ k=1 \]

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccc}
3 & 6 & 9 & 12 \\
\end{array}
\]
1-d 1-stencil: shared memory

Global memory

Shared memory \( s[] \)

Global memory

Read

\_\_syncthreads()

Compute

Write output

\[
\begin{array}{c|ccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
\hline
T0 & T1 & T2 & T3 \\
\hline
3 & 6 & 9 & 12 \\
\end{array}
\]
1-d 1-stencil: shared memory

Global memory

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

Shared memory

s[]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

Global memory

<table>
<thead>
<tr>
<th>3</th>
<th>6</th>
<th>9</th>
<th>12</th>
</tr>
</thead>
</table>

Read

__syncthreads() Compute

Goal: eliminate shared memory access
1. Determine warp input

assume 4 threads/warp

<table>
<thead>
<tr>
<th>Global memory input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
1. Determine warp input

assume 4 threads/warp

Global memory input

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Warp 0

Warp 1
2. Assign input to *owner* thread

<table>
<thead>
<tr>
<th>Global memory</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. Assign input to owner thread

Global memory

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

rc
2. Assign input to *owner* thread

Global memory

Distribute cyclic distribution

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td>[5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some thread inputs are remote!

Global memory

Distribute rc


[0]+[1]+[2]

Not available!
We define new communication primitives

- **Receive**(src_tid, remote_reg) – receive data stored in thread src_tid in remote variable remote_reg
- **Publish**(local_reg) – publish local data stored in variable local_reg
- For one thread to Receive, another has to Publish!
2. Communication phase: Receive

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td>[4]</td>
<td>[5]</td>
<td></td>
</tr>
</tbody>
</table>

\[(0)+[1]+[2]\]
2. Communication phase: Receive

<table>
<thead>
<tr>
<th>rc</th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
</tr>
<tr>
<td></td>
<td>[4]</td>
<td>[5]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Receive (src,what) \(v = R(T0, rc[0])\)

\([0] + [1] + [2]\)
2. Communication phase: Publish

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
</tr>
</tbody>
</table>

Receive

Publish

\[ v = R(T_0, rc[0]) \]

\[ P(rc[0]) \]

\[ [0] + [1] + [2] \]
## 2. Communication phase: Publish

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Receive</th>
<th>Publish</th>
<th>Receive</th>
<th>Publish</th>
<th>Receive</th>
<th>Publish</th>
<th>Receive</th>
<th>Publish</th>
</tr>
</thead>
<tbody>
<tr>
<td>v=R(T0, rc[0])</td>
<td>P(rc[0])</td>
<td>v=R(T1, rc[0])</td>
<td>P(rc[0])</td>
<td>v=R(T2, rc[0])</td>
<td>P(rc[0])</td>
<td>v=R(T3, rc[0])</td>
<td>P(rc[0])</td>
</tr>
</tbody>
</table>
## 3. Computation phase

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Receive (R)</th>
<th>v=R(T0,rc[0])</th>
<th>v=R(T1,rc[0])</th>
<th>v=R(T2,rc[0])</th>
<th>v=R(T3,rc[0])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publish (P)</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
</tr>
<tr>
<td>Compute</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
</tr>
</tbody>
</table>

_ac=[0], need [1]
2. Communication phase: Receive

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td></td>
<td>[0] [4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive (R)</td>
<td>v=R(T0,rc[0])</td>
<td>v=R(T1,rc[0])</td>
<td>v=R(T2,rc[0])</td>
<td>v=R(T3,rc[0])</td>
</tr>
<tr>
<td>Publish (P)</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
</tr>
<tr>
<td>Compute</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
</tr>
<tr>
<td>Receive</td>
<td>v=R(T1,rc[0])</td>
<td>v=R(T2,rc[0])</td>
<td>v=R(T3,rc[0])</td>
<td>v=R(T0,rc[1])</td>
</tr>
</tbody>
</table>
## 2. Communication phase: Publish

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
</tr>
<tr>
<td>Receive (R)</td>
<td>v=R(T0,rc[0])</td>
<td>v=R(T1,rc[0])</td>
<td>v=R(T2,rc[0])</td>
<td>v=R(T3,rc[0])</td>
</tr>
<tr>
<td>Publish (P)</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
</tr>
<tr>
<td>Compute</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
</tr>
<tr>
<td>Receive</td>
<td>v=R(T1,rc[0])</td>
<td>v=R(T2,rc[0])</td>
<td>v=R(T3,rc[0])</td>
<td>v=R(T0,rc[1])</td>
</tr>
<tr>
<td>Publish</td>
<td>P(rc[1])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
</tr>
</tbody>
</table>
### 3. Computation phase

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>[0][4]</td>
<td>[1][5]</td>
<td>[2]</td>
<td>[3]</td>
</tr>
<tr>
<td><strong>Receive (R)</strong></td>
<td>v=R(T0,rc[0])</td>
<td>v=R(T1,rc[0])</td>
<td>v=R(T2,rc[0])</td>
<td>v=R(T3,rc[0])</td>
</tr>
<tr>
<td><strong>Publish (P)</strong></td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
</tr>
<tr>
<td><strong>Compute</strong></td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
</tr>
<tr>
<td><strong>Receive</strong></td>
<td>v=R(T1,rc[0])</td>
<td>v=R(T2,rc[0])</td>
<td>v=R(T3,rc[0])</td>
<td>v=R(T0,rc[1])</td>
</tr>
<tr>
<td><strong>Publish</strong></td>
<td>P(rc[1])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
</tr>
<tr>
<td><strong>Compute</strong></td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
</tr>
</tbody>
</table>

_ac=[0]+[1], need [2]
### 4. write result to global memory

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rc</strong></td>
<td>[0] [4]</td>
<td>[1] [5]</td>
<td>[2]</td>
<td>[3]</td>
</tr>
<tr>
<td><strong>Receive (R)</strong></td>
<td>v=R(T0,rc[0])</td>
<td>v=R(T1,rc[0])</td>
<td>v=R(T2,rc[0])</td>
<td>v=R(T3,rc[0])</td>
</tr>
<tr>
<td><strong>Publish (P)</strong></td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
</tr>
<tr>
<td><strong>Compute</strong></td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
</tr>
<tr>
<td><strong>Receive</strong></td>
<td>v=R(T1,rc[0])</td>
<td>v=R(T2,rc[0])</td>
<td>v=R(T3,rc[0])</td>
<td>v=R(T0,rc[1])</td>
</tr>
<tr>
<td><strong>Publish</strong></td>
<td>P(rc[1])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
</tr>
<tr>
<td><strong>Compute</strong></td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
</tr>
<tr>
<td><strong>Receive</strong></td>
<td>v=R(T2,rc[0])</td>
<td>v=R(T3,rc[0])</td>
<td>v=R(T0,rc[1])</td>
<td>v=R(T1,rc[1])</td>
</tr>
<tr>
<td><strong>Publish</strong></td>
<td>P(rc[1])</td>
<td>P(rc[1])</td>
<td>P(rc[0])</td>
<td>P(rc[0])</td>
</tr>
<tr>
<td><strong>Compute</strong></td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
<td>_ac+=v</td>
</tr>
</tbody>
</table>

_\text{ac}=[0]+[1]+[2]_
Receive + Publish = \textit{shuffle}

<table>
<thead>
<tr>
<th>Receive (R)</th>
<th>Publish (P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v = R(T0, rc[0])</td>
<td>P(rc[0])</td>
</tr>
<tr>
<td>v = R(T1, rc[0])</td>
<td>P(rc[1])</td>
</tr>
<tr>
<td>v = R(T2, rc[0])</td>
<td>P(rc[1])</td>
</tr>
</tbody>
</table>
Receive + Publish = shuffle

Receive (R) Publish (P)

<table>
<thead>
<tr>
<th>Receive</th>
<th>Publish</th>
</tr>
</thead>
<tbody>
<tr>
<td>v=R(T0,rc[0])</td>
<td>P(rc[0])</td>
</tr>
<tr>
<td>v=R(T1,rc[0])</td>
<td>P(rc[1])</td>
</tr>
<tr>
<td>v=R(T2,rc[0])</td>
<td>P(rc[1])</td>
</tr>
</tbody>
</table>

pub_idx=0; src=0;
v=shuffle(src, rc[pub_idx])

pub_idx=1; src=1;
v=shuffle(src, rc[pub_idx])

pub_idx=1; src=0;
v=shuffle(src, rc[pub_idx])
Performance benefits for k-stencil

Up to 76%!

See the paper for further analysis for benefits and limitations
Summary: Register Cache

• Start from shared memory-based implementation
• Identify input for each warp
• Distribute data among threads
• Split in multiple phases
  – Communication phase: Publish – Receive
  – Computation phase
• Transform Publish-Receive into *shuffle*
Part 2: multiplication in large binary fields $2^n$ with $32 < n < 256$

- Binary field multiplication – computational bottleneck in many applications
  - Security, Storage

- Typical scenario: multiply many pairs

- Main kernel: convolution of binary vectors of size $n$

- x86 CPUs: special CLMUL instruction
  - IvyBridge: 14 cycles, 2 convolutions
Binary convolution

\[
\begin{array}{c}
\text{Input} \\
\begin{array}{c}
\text{v}_1 \\
1 \ 0 \ 1
\end{array} & \begin{array}{c}
\text{v}_2 \\
1 \ 1 \ 1
\end{array}
\end{array}
\]
Binary convolution

\[ v_1 = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \]

\[ v_2 = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \]

\[ \& \]

\[ \begin{bmatrix} 1 \end{bmatrix} \]
Binary convolution

\[ v_1 \]

\[ \begin{array}{ccc}
1 & 0 & 1 \\
\& & \& \\
1 & 1 & 1
\end{array} \]

\[ v_2 \]

\[ \begin{array}{ccc}
1 & 1 & 1 \\
\& \XOR \\
1 & 1
\end{array} \]
Binary convolution

\[ v_1 \]

\[ \begin{array}{ccc} 1 & 0 & 1 \\ & \& & \& \\ 1 & 1 & 1 \\ \end{array} \]

\[ v_2 \]

\[ \begin{array}{ccc} 1 & 1 & 0 \\ \end{array} \]
Binary convolution

\[ v_1 \]

\[ 1 \ 0 \ 1 \]

&

&

\[ v_2 \]

\[ 1 \ 1 \ 1 \]

\[ \text{XOR} \]

\[ 1 \ 1 \ 0 \ 1 \]
Binary convolution

$v_1$  
1 0 1

$\&$

$v_2$
1 1 1

1 1 0 1 1
Challenges - Solutions

- Bit-level operations

- Load balancing between warp threads

- Scaling to large fields
Challenges - Solutions

- Bit-level operations
- Load balancing between warp threads
- Scaling to large fields

- Bit slicing
  Compute 32 convolutions in a single thread
- Algorithmic trick to achieve divergent free execution
- Use register cache to free shared memory and scale better

See the paper for details
Performance

- CPU baseline: **CLMUL intrinsic** (via popular NTL library)
- NVIDIA K80: 138x faster than CPU

![Graph showing speedup vs field size]
Performance

- CPU baseline: **CLMUL intrinsic** (via popular NTL library)
- NVIDIA K80: 138x faster than CPU
Conclusions

- Register cache: general technique for replacing shared memory with shuffle
- Apply to fast binary field multiplication
- Register cache improved application performance by 50%
- Total: x138 over CPU CLMUL for fields of size 32

Source code: https://github.com/HamilM/GpuBinFieldMult

Further questions: mark@ee.technion.ac.il