Mapping Data-intensive Applications to an Explicitly Managed Memory Architecture: Challenges and Solutions

Pierre G. Paulin
Director, SoC Platform Automation
STMicroelectronics (Canada) Inc.

Memory Architecture and Organization Workshop
Montreal, Canada, 3 October 2013
Outline

• Cache vs. Explicitly Managed Memory (EMM)

• STHORM (aka Platform 2012) many core fabric
  • Explicitly Managed Memory (EMM) architecture

• STHORM programming tools

• Computer vision application benchmark results

• Challenges of programming EMM architectures

• Outlook: Emerging programming tool solutions
Cache vs. EMM

- Core
- Core
- Core
- Local interconnect
- Shared L1 TCDM
- Coherent L2 $
- Coherency H/W
- L3 Memory
- Refill H/W
- DMA
- L2 [Memory | $]
- DMA
- L2 Memory
- DMA
- L3 Memory

1 cycle
10's cycles
100's cycles
Cache vs. Explicitly Managed Memory

• Cache
  ✓ High performance
  ✓ Programming simplicity
  ✗ Higher power, cost
  ✗ Unpredictable performance

• EMM
  ✓ Lower power, cost
  ✓ Performance predictability
  ✓ Explicit control of data movement
  ✗ Poor handling of irregular accesses (scatter-gather)
  ✗ Programming complexity
  • Use of DMAs for explicit data movement
  • Need to overlap communication and computation
STHORM Many-core Fabric: Explicitly Managed Memory Architecture
STMicroelectronics STHORM Platform

GOPS/mm² – GOPS/W

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>8</th>
<th>&gt; 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>General-purpose Computing</td>
<td>Throughput Computing</td>
<td>Mixed</td>
<td>HW IP</td>
</tr>
<tr>
<td>CPU</td>
<td>GP-GPU</td>
<td>SW</td>
<td>STHORM</td>
</tr>
</tbody>
</table>

- Throughput Computing (GP-GPU) and SW work together to improve performance across various applications.
- STHORM integrates with HW and SW to enhance overall system efficiency.
- HW IP represents the hardware intellectual property used to augment and optimize performance.
What is STHORM?

- Massively parallel **programmable** processor:
  - Delivers high performance per mm$^2$  $\rightarrow$  20 GOPS/cluster
    $\rightarrow$  3.7 mm$^2$/cluster (28nm)
  - Low-power: can be battery-powered  $\rightarrow$  100 GOPS/W
  - Explicitly managed memory architecture
  - Scalable GALS architecture: each cluster has its own F,V point
STHORM Cluster Architecture

- Scalable GALS architecture
- Each cluster has its own operating point (F,V)
- Asynchronous 2D mesh interconnect
LD/ST and DMA Memory Transfers

- NUMA architecture
- Intra-Cluster:
  - LD/ST (UMA)
  - DMA: From/to TCDM
- Inter-Cluster:
  - LD/ST (NUMA)
  - DMA: L1-to/from-L1
- Cluster to/from L2-Mem:
  - LD/ST (NUMA)
  - DMA: L1 to/from L2
- Cluster to/from L3-Mem (though the system bridge):
  - LD/ST (NUMA)
  - DMA: L1 to/from L3
STORM Cluster Architecture

- Parametric multi-core crossbar with a logarithmic structure
- Reduced arbitration complexity
- Round robin arbitration scheme
- Up to N memory accesses per cycle
- Test-and-Set support
Shared Memory

N Processes

Routing Tree

Arbitration Tree

2N Memory Banks

P1 P2 P3 P4

B1 B2 B3 B4 B5 B6 B7 B8
Shared Memory Conflicts

Theoretical Measured

Computation Time

0% 35.70% 50% 66.70% 100%

0% 35.70% 50% 66.70% 100%

0 0.2 0.4 0.6 0.8 1 1.2 1.4

Shared Memory Conflicts

Typical case
• Supports 1D & 2D transfers
• Up to 3.2GB/s peak per DMA
• Support up to 16 outstanding transactions
• Support of Out of Order (OoO)
STHORM Programming Tools
Programming STHORM

Mapping control
Mapping info

Application
Prog. Model

Performance Feedback & debug

Host
Fabric

 ARM Cortex A9
DMA
FC Mem
Fabric Ctrl

Ctrl Periph.
DMA
FC Mem
Fabric Ctrl

HWPE 0
... 
HWPE N

Send Recv Rd Wr A-LIC

HW Synchr.
Shared L1 MEM

Ctrl Peri.
Clus. Ctrl
PE 0 PE 1 ...
PE n

Async NoC

Fabric Host

System bus
OpenCL: Standard of the Convergence

Task Parallelism (run-to-completion)

Data Parallelism (with some-synchro)

OpenCL

Cortex-A9
Cortex-A15

STHORM (aka P2012)

Midgard-T6xx Rogue

More parallelism

More programmability

Multi-core CPU

Many-core

GPU
STHORM as OpenCL Device

Conceptual OpenCL Device

- Scalable programming model
- Supports SPMD model
- Supports Task parallelism
- Covers complex memory hierarchy
- Supports async memory transfers

STHORM Compute Device

- Scalable architecture (cluster based)
- Supports SPMD with 0-cost branch divergence
- Supports Task parallelism
- Shared Local memory
- 1D/2D DMA engines
- Hardware synchronizer
Global OpenCL Execution Model

- Host-centric, devices are asynchronous
- Supports two kind of parallelism
  - DLP (Data Level Parallelism): Parallel multiple instances of a kernel
  - TLP (Task Level Parallelism): Different kernels executed in parallel
- Communication
  - Between work-items of same workgroup
  - Between kernels: through buffers in global memory
  - Between host and kernels: memory mapped buffers
- Synchronization
  - Between work-items: work-group ‘barrier’
  - Between commands of a command queue: out-of-order
STHORM ≠ GPU: Kernel Level Parallelism

Independent Clusters
+ Independent Processors
  - Work-item divergence not an issue in Sthorm
  - GPUs can support, but cost of stalling threads
  - STHORM supports more complex OpenCL task graph than GPUs. Both **task-level** and **data-level (ND-Range)** are possible

STHORM cores are not HW-multithreaded
  - STHORM OCL runtime does not accept more than 16 work-items per work-group when creating an ND-Range.
  - But you can choose which work to do in each work-item (e.g. using “case”)
STHORM Memory Mapping

L3
Ext
mem

Host memory
(virtualized)

Global Memory
(Physically contiguous)

1 MB

Program cache refill, allocation of static & non-static buffers

~2 cycles

L2
1 MB

256 KB

Local Memory
Private (work-
item stacks)

~2 cycles

Constant Memory

Runtime
~10 KB

resident

non-resident
(per kernel run)

Semi-resident
(per program build)
Data Movements inside STHORM

L3
Global Memory (buffers)

Scalar/Vector load/store
async_work_group_copy

async_work_group_2d_copy
async_work_item_[2d]_copy

L1
Local Memory
Private (work-item stacks)
Constant Memory

Runtime
~10 KB
OpenCL Programming Tools

OpenCL Encodings
- Symbolic names / physical IDs
- Trace point states (enabled / disabled)
- Compact encoding to reduce instrumentation cost
- Debug symbolic info

OpenCL-aware visualization & analysis tools

Low-level Traces

Trace Database

OpenCL-aware debugger
STHORM Target Platform
STHORM Test Chip

- 28nm LP CMOS process
- 4 Clusters, 69 dual issue processors
- 80 GOPS
- 1MB L2 mem
- 600 MHz typ
- 200mW per cluster typical (400 mW worst case)
- 3.7 mm² per cluster
- Fully functional samples available since Dec. 2012

Energy efficiency = 100 GOPS / W
STHORM Evaluation Board

- Ethernet
- 2 CMOS imaging plug-in
- Zynq host: Cortex A9, Peripherals
- STHORM test chip
- MEMS: acc./gyro/magneto; mikes (2x)
- 3x USB (OTG, serial, JTAG)
- Bluetooth module
- Debug & Trace
Benchmarking Visual Analytics Applications

- Objective: Evaluation of achievable speedup when offloading visual analytics applications from ARM host to STHORM accelerator
  - Corner detection: FAST, SIFT
  - Feature tracking: PKLT
- Input image: VGA resolution, 777 corners detected
- Measurement results collected by analyzing execution traces from STHORM evaluation board
Visual Analytics Benchmark

Running Time (ms)

- KLT (No Pyramid)
- PKLT (1 Level)
- FAST9
- FAST ROSTEN

STHORM - 16 proc. 600 Mhz
ARM + Neon Dual Core. 1GHz

Speed ST HORM 1 Cluster Versus 1 CA9

- KLT (No Pyramid)
- PKLT (1 Level)
- FAST9
- FAST ROSTEN
- Geomean

ARM Cortex A9 Dual Core (ONE used)

1 Cluster STHORM (3.7 mm²)
STHORM 4 Clusters
Visual Analytics Benchmark

Running Time (ms)

- KLT (No Pyramid)
- PKLT (1 Level)
- FAST9
- FAST ROSTEN

- STHORM - 16 proc. 600 Mhz
- ARM + Neon Dual Core. 1GHz

8X GOPS/mm²
25X GOPS/mm²*W

ARM Cortex A9 Dual Core (ONE used)

1 Cluster STHORM (3.7 mm²)

STHORM 4 Clusters
KLT Scalability: Execution Time

Same compiled code, Different runtime parameters (ND-Range)

Execution Time (ms)
(less is better)

<table>
<thead>
<tr>
<th>System</th>
<th>1 cluster</th>
<th>2 clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>STHORM 4 Proc - 600 Mhz</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>STHORM 8 Proc - 600 Mhz</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>STHORM 16 Proc - 600 Mhz</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>SHORM 32 Proc - 600 Mhz</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CA9 1GHz</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>

WG = work-group, WI = work-item
KLT Scalability: Speed-up

Same compiled code, Different runtime parameters (ND-Range)

Speedup vs CA9
(more is better)

1 cluster
(1 WG, <n> WI)

1.8x

3.5x

6x

2 clusters
(2 WG, 32 WI)

10.5x

STHORM 4 Proc - 600 Mhz
STHORM 8 Proc - 600 Mhz
STHORM 16 Proc - 600 Mhz
STHORM 32 Proc - 600 Mhz

WG=work-group,
WI=work-item
FAST Application Performance

FAST Running time (ms) detection Kernel - VGA Image - 777 Keypoints

<table>
<thead>
<tr>
<th>System</th>
<th>Time (ms)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM CA9 dual-core sequential version - 1 GHz</td>
<td>260</td>
<td></td>
</tr>
<tr>
<td>Intel i5 - 2.5 GHz</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>STHORM16 cores - 600 MHz</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Quadro NVS3100M - 1.5 GHz</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>STHORM 4 clusters 64 cores - 600 MHz</td>
<td>7</td>
<td>37x</td>
</tr>
<tr>
<td>Nvidia GeForce GTX 280 (240 Processors) - 1.5 GHz</td>
<td>1.3</td>
<td></td>
</tr>
</tbody>
</table>

Best in class performance
FAST Application Power

Frame/sec/W - VGA Image - 777 Keypoints

- STHORM16 cores - 600 MHz
- Quadro NVS3100M - 1.5 GHz
- STHORM 4 clusters 64 cores - 600 MHz
- Nvidia GeForce GTX 280 (240 Processors) - 1.5 GHz

$\times40$, $\times20$

Best in class energy efficiency
Challenges of Programming EMM Architectures
OpenCL Trace Visualization Ex. (1 Cluster)
OpenCL Trace Visualization Ex. (4 cluster)
Main Source of Complexity and Bugs

Data **Slicing/Tiling** for exploiting the L1 memory

- **Data Slicing/Tiling**
  - work-group
  - Tile 0
  - Tile 1
  - 16 work-items

Asynchronous transfer operations
To fully exploit the DMA

**Merge multiple functions into a single OpenCL kernel**
- to maximize the data locality and reduce host-device interaction

**Sobel 3x3 performance**
- (lower is better)
  - 1 OpenCL kernel: 100
  - 3 OpenCL kernels: 378

**Sobel**
- Gradient x
- Gradient y
- Euclidean norm
Hide Memory Latency: Software Pipeline

Kernel code

Work-group async-copy \underline{Next} input data to local
Work-group async-copy \underline{Previous} output data to global
Process my own \underline{Current} input local block
Work-group barrier
Wait for async-copies completion
Hide Memory Latency: Double Buffer

Local memory

Prolog

Loop body

Epilog

```
Read data  
block 0 → local in0

Compute data  
local in0 → local out0

Write data  
local out0 → block 0

Read data  
block 1 → local in1

Compute data  
local in1 → local out1

Read data  
block 2 → local in0

Write data  
local out i%2 → block i-2

Write data  
local out (i+1)%2 → block N-2

Write data  
local out N%2 → block N-1

Write data  
local out (N+1)%2 → block N-1

Read data  
block i → local out i%2

Write data  
local out (N+1)%2 → block N-2

Read data  
block (N+1)%2 → local out (N+1)%2
```
OpenCL Pros

• OpenCL enables full performance of STHORM EMM
  • Explicitly managed memory by exploiting DMAs and L1 memory
  • OpenCL can adapt to different Host-Fabric system configurations

• OpenCL well-suited to image processing applications
  • Exploiting data-level parallelism is enough in most cases
  • Still possibility of coarse grain task parallelism with kernel graphs

• Conceptually simple execution model
  • WG barriers
  • Atomic operations

• Good scalability
  • One kernel, multiple execution configurations (nb of clusters, of PE)
OpenCL Cons

• Lower level programming
  • Low level programming (host programming and kernel memory management)
  • First parallelization steps are error-prone (C → OpenCL, image tiling)
  ➢ Substantial Learning curve, need knowledge of target architecture
  ➢ Specialization of code for the target
  ➢ Lower programming productivity
  ➢ Wide range of results, depending on programmer experience

➢ Performance and bandwidth issue with low compute intensive kernels
  • Data round trip between L1 and L3
  • Host-device interaction
  ➢ Need to manually merge logical functions into single kernel
Outlook: Emerging Programming Tool Solutions
STHORM Programming Outlook

Expert programmer
API + OpenCL-C

Algorithm designer
Computer Vision
Specific Language

KernelGenius
OpenCL-C Kernel
C host wrapper

SThorm
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Kernel Genius Inputs

Graph

NxM Image

R: [-2:2] [0:0]

W: [0:0] [0:0]

5x1 convol

W: [0:0] [0:0]

RxM Image

R: [-2:2] [0:0]

1x5 convol

W: [0:0] [0:0]

NxM Image

Read/Write patterns

replication pattern

accessx

[-1:1][-1:1]

[0:3][0:3]

[-1:3][-1:2]

stridex

stridex

data set
Read/Write Pattern and Stride Examples

- **(a)** 5x5 Blur Node
  - read_x = [-2:2]
  - stride_in_x = 1
  - write_x = [0:0]
  - stride_out_x = 2

- **(b)** 3x3 Gradient Node
  - read_x = [-1:1]
  - stride_in_x = 1
  - write_x = [0:0]
  - stride_out_x = 2

- **(c)** 4x4 IDCT Node
  - read_x = [0:3]
  - stride_in_x = 4
  - write_x = [0:4]
  - stride_out_x = 4

- Read Pattern
  - read_Y = [-2:2]
  - stride_in_Y = 1

- Write Pattern
  - write_Y = [0:4]
KernelGenius Compiler

- Graph merge
- Data tiling
- Tile based scheduling
- Size the local buffering
- Parallelize the processing
- Manage Image Borders
- Generate the OpenCL Kernel + C wrapper
Buffer Calculation Examples

(a)

(b)
Buffer Management and Scheduling

A - Scheduled and buffer allocated graph

B - Scheduler loop iterations

C - 5x5 blur
D - 3x3 gradient

Cycle 0
Rate 2

Cycle 1
Rate 1
B-Down sample

Cycle 3
Rate 1
C-5x5 blur

Cycle 4
Rate 1
D-3x3 gradient

Cycle 5
Rate 1
E-Copy tile out

Async operation

Synchronous processing

Pop=1 Peek range : [0:0] Push=1
width

Pop=1 Peek range : [-1:1] Push=1
width/2

Pop=1 Peek range : [-2:2] Push=1

Peek range : [0:0] Push=1
width/2

Peek range : [-1:1] Push=1
width/2

Peek range : [-1:1] Push=1
width

Peek range : [0:0]

4 tiles

5 tiles

3 tiles

2 tiles

0 1 2 3 4 5 \( \ldots \) N N+1 N+2 N+3 N+4 N+5

2A_i 2A_i 2A_i 2A_i 2A_i
e_i e_i e_i e_i e_i
t_i t_i t_i t_i t_i

B B B B

C C C C C

D D D D D

2A_w 2A_w 2A_w 2A_w 2A_w

E_w E_w E_w E_w E_w

Prolog Steady state Epilog
C code

```c
void ImageDiff(int w, int h,
    float *in1,
    float *in2) {
for (y=0;y<h;y++) {
    for (x=0;x<w;x++) {
        out[y*w+x]=in1[y*w+x]-in2[y*w+x]
    }
}
}
```

KernelGenius code

```c
kernel ImageDiff(int w, int h,
    float in1[h][w],
    float in2[h][w]) {
    Operator<float> sub(in1,in2) {
        .function=${ @sub=$in1-$in2; }$
    }
    return sub;
}
```

OpenCL kernel code

```c
__kernel void ImageDifference(__global DATA_TYPE *global_output,
    __global DATA_TYPE *global_input0,
    __global DATA_TYPE *global_input1,
    __local  DATA_TYPE *local_buffer0,
    __local  DATA_TYPE *local_buffer1,
    __local  DATA_TYPE *local_buffer2, int image_width, int image_height) {

    event_t e_in[2], e_out;
    int id = get_local_id(0),nb_it = get_local_size(0);

    local DATA_TYPE *local_in0[2] = {local_buffer0, local_buffer0+image_width};
    local DATA_TYPE *local_in1[2] = {local_buffer1, local_buffer1+image_width};
    local DATA_TYPE *local_out[2] = {local_buffer2, local_buffer2+image_width};

    for (int i=0;i<(image_height+2);i++) {
        if (i>1) {
            e_out= async_work_group_copy(global_output, local_out[i&0x1], image_width, 0);
            global_output+=image_width;
        }
        if (i<image_height) {
            e_in[0]= async_work_group_copy(local_in0[i&0x1], global_input0, image_width, 0);
            global_input0+=image_width;
            e_in[1]= async_work_group_copy(local_in1[i&0x1], global_input1, image_width, 0);
            global_input1+=image_width;
        }
        if ((i>0)&&(i<=image_height)) {
            int j=(~i)&0x1; for (int k=id;k<image_width;k+=nb_it) {
                local_out[j][k]= local_in0[j] [k] - local_in1[j] [k];
            }
            barrier(CLK_LOCAL_MEM_FENCE);
        }
        if (i>1) { wait_group_events(1, &e_out); }
        if (i<image_height) { wait_group_events(2, (event_t *)&e_in); }
    }
}
```
Kernel Genius early results (1)

A- 3x3 Sobel filter, 1080p

B- 3x3 Gradient filter, 1080p
Kernel Genius early results (2)

**1-Fixed coefficients**

- Manual OpenCL kernel: 100
- KernelGenius: Graph with 2 'Filter' nodes: 99
- Manual OpenCL kernel: 124
- KernelGenius: Graph with 2 'Filter' nodes: 122

**2-Generic coefficients**

*C- 5x5 2D Separable Convolution, 1080p*
Canny Edge Example (Step 1)

Step 1: OpenCL Kernel
- Noise reduction
- Gradient
- Non Maxima Suppression

Step 2: C code (Host)
- Edge Points Connection
- Thinning Edges

Step 3: OpenCL Kernel

**Canny Edge Process:***
1. **Step 1:**
   - **Noise reduction**:
   - **Gradient**:
     - Gradient X
       - Gaussian Blur
       - X Blur
       - Y Blur
     - Gradient Y
       - 7x1 convol
       - 1x7 convol
       - 7x1 convol
       - 1x7 convol
   - **Non Maxima Suppression**:

2. **Step 2:**
   - Edge Points Connection
   - Thinning Edges

3. **Step 3:**
   - OpenCL Kernel

**Nodes:**
- Convolution predefined parametric nodes
- Operator programmable node
- Filter programmable node
- Operator programmable nodes
Canny Edge Detector Result

![Bar Chart]

- Manual OpenCL kernel: 100
- KernelGenius Generated Version: 90.6

Lower is better
Kernel Genius Lessons

• Use of domain-specific progr. model on EMM

• Pros
  • Higher-level of abstraction
  • Automation of DMA-based data movement
  • Automation of scheduler generation
  • Automation of buffer sizing

• Cons
  • Domain-specific: Imaging and Computer Vision
  • Does not cover some highly dynamic use cases
    • Revert back to OpenCL
  • Non-standard programming model
STHORM Programming Outlook

Expert programmer
API + OpenCL-C

Algorithm designer
Computer Vision Specific Language

Computer Vision standard API

OpenCL-C Kernel
C Host wrapper

OpenVX

KernelGenius

Cluster 1
Cluster 2
Cluster 3
Cluster 4
Conclusion

• Explicitly managed memory architecture leads to high-performance at low-cost and power
  • Over 35X speedup wrt ARM Cortex A9 @1GHz for typical visual analytics applications
  • 20X to 40X power reduction wrt GPUs

• Development with OpenCL allows to fully exploit the architecture \(\rightarrow\) at the expense of extra development effort
  • Double buffering
  • Management of DMA-based data movement
  • Kernel fusion

• Programming tools outlook
  • Domain-specific higher-level capture and automation yield best time-to-market and similar performance to OpenCL-based capture
Thank you!
Questions?