ePUMA
embedded Parallel DSP processor with Unique Memory Access

ePUMA team

Division of Computer Engineering

December 7, 2011
1 Introduction

2 ePUMA overview

3 Sleipnir

4 ePUMA toolchain

5 ePUMA programming

6 Case study - Discrete Fourier Transform (DFT)
Introduction

The ePUMA team:

- Dake Liu
- Jian Wang
- Joar Sohl
- Andréas Karlsson
- Magnus Pettersson
- Andreas Ehliar
Applications

Originally in ePUMA we considered the following applications:

- Baseband processing.
- Media processing.
- Radar.
Properties of DSP algorithms

As most DSP algorithms share some common traits, in ePUMA we optimized for

- Predictable addressing. I.e the addresses of the accessed values are not data dependant.
- Few branches other than back jumps in loops.
- Constant iteration counts.
1 Introduction

2 ePUMA overview

3 Sleipnir

4 ePUMA toolchain

5 ePUMA programming

6 Case study - Discrete Fourier Transform (DFT)
Highly parallel processor for predictable DSP tasks

Heterogenous design:
- 1 master control processor
- 8 slave processor cores

Exploited parallelism:
- Task-parallelism (several processor cores)
- Data-parallelism (SIMD instructions on slave processors)
System overview

- Sleipnir 0
- Sleipnir 1
- Sleipnir 2
- Sleipnir 3
- Sleipnir 4
- Sleipnir 5
- Sleipnir 6
- Sleipnir 7

Master DMA
Main Memory
Memory hierarchy

Off chip main memory

On chip interconnection

Master LS

Sleipnir 0 LS

Sleipnir 7 LS

Level 1

Level 2

Level 3

Master Core

Sleipnir Core

Registers

PM

DM 0

DM 1

PM

CM

LVM 1

LVM 2

LVM 3

PM

CM

LVM 1

LVM 2

LVM 3

PM

CM

LVM 1

LVM 2

LVM 3
Sleipnir Local Store

Diagram showing the integration of DMA, NoC, Switch 1, Switch 2, and LVMs (LVM 1, LVM 2, LVM 3) into the Sleipnir Local Store and Core.
Introduction

ePUMA overview

Sleipnir

ePUMA toolchain

ePUMA programming

Case study - Discrete Fourier Transform (DFT)
Sleipnir features

- Scratchpad memory based programming - no data cache
- Up to 16-way SIMD datapath (operates on 128 bit data vectors)
- Up to 16 real or 4 complex multiplications per cycle (16 bit data)
- Supported datatypes:
  - Real fixed-point data: 8, 16, 32 bits
  - Complex fixed-point data: 16, 32 bit real and imaginary parts
  - Single precision floating-point (32 bits)
- Special purpose instructions: DCT, butterflies, vector sort...
Sleipnir features cont.

- Hardware repeat - overhead-free loops
- Loop registers - for faster software loops
- Instruction repeat - simplify programming / high code density
- Predicated (condition based) execution - Reduces branches
- Multi-bank local vector memory with permutation - conflict-free parallel memory access
- Rich set of addressing modes - flexibility!
Sleipnir customization

Many parameters can be customized:

- Instruction set
- Local memory and register file sizes
- AGU capabilities
- Accelerators

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local vector memory (LVM) size</td>
<td>Up to 8k 128-bit vectors (128kB)</td>
</tr>
<tr>
<td>Register file size</td>
<td>8-32 vectors (0.125 - 0.5 kB)</td>
</tr>
<tr>
<td>Constant memory size</td>
<td>Up to 256 vectors (4kB)</td>
</tr>
<tr>
<td>Program memory size</td>
<td>Typically 8-16 kB</td>
</tr>
</tbody>
</table>
Addressing

Normally many cycles are wasted on rearranging data with shuffle-instructions. This is often due to issues with data alignment and bank-conflicts.
Data access

Consider the following address layout in a single bank memory. The only vectors of length four that can be accessed in one cycle is the row vectors \{0,\ldots,3\}, \{4,\ldots,7\}, \{8,\ldots,11\} and \{12,\ldots,15\}. Accessing one of the colored column vectors take 4 cycles.

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
\end{array}
\]
Multi-bank

By splitting the memory into different banks (which increases the area cost somewhat), the only constraint is that no two elements reside in the same bank. So while we may now access e.g. vectors \{x, \ldots, x+3\} in one cycle, the columns still take four cycles to access.
Multi-bank and permutation

Given that the access patterns are known in advance, as is common in DSP algorithms, we may reorder the physical addresses of the logical addresses. An example of a permutation that allows single cycle access for the columns can be seen below. No two elements of any column reside in the same memory bank.

<table>
<thead>
<tr>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>12</td>
</tr>
</tbody>
</table>
Instruction set overview

Simple instructions:
- Vector addition/subtract/min/max/absolute
- Vector logical
- Vector shift
- Data type conversions

Special purpose instructions:
- Complex butterflies - 2 radix-2 or 1 radix-4 / cycle
- Vector sort - sort 4 elements (2 in parallel)
- Vector sort - sort 8 elements (if half-vector presorted)
- Median - of 3 elements (2 in parallel)
- Discrete Cosine Transform
- Complex absolute squared maximum/minimum search
- Byte level sum of absolute differences (motion estimation)
1 Introduction

2 ePUMA overview

3 Sleipnir

4 ePUMA toolchain

5 ePUMA programming

6 Case study - Discrete Fourier Transform (DFT)
The ePUMA programming model is kernel based. Application code is written as normal and the Sleipnir cores are accessed by calling kernels.

We divide developers in two categories:

- Application developers that uses ePUMA by kernel library.
- Kernel developers.
Overview of expected toolchain.

- C-program
  - C-compiler
    - Senior asm
      - Assemblers/Simulator/Hardware
  - Kernel calls
    - Control & DMA
    - Sleipnir asm
  - Select & configure
A kernel consists of three parts.

- DMA & control specification or code for Senior and Sleipnir.
- Sleipnir assembly code.
- Kernel specification. This is used to match kernel calls with specific kernels and generate the call specific parameters.
Select kernel

For each kernel call the parameters are matched with the kernel specifications in the kernel library, and the most suitable kernel is selected. E.g. depending on if the input data fits in the LVMs or not different implementations might be used.
Configure kernel

Kernel configuration:
- Configure code.
- Generate permutation vectors.
- Schedule the computation.
Schedule

- DMA-C
- DMA-Spec
- C-compiler
- Sleipnir asm
- Senior asm
- Sleipnir asm
- Scheduler
As ePUMA has been designed for predictable computing, the scheduling is static. For a kernel to be schedulable by the scheduler, it must be represented as a DAG.
Task

A task behaves as a function; it consumes input data and produces output data. It does not maintain any state. The following interface needs to be specified for each task.

- Input set of data blocks in memory 0.
- Input set of data blocks in memory 1.
- Output set of data blocks in memory 0.
- Output set of data blocks in memory 1.
Suggestions for first steps.

- Kernel based. The kernels are presented as actors.
- Compile stateless data parallel parts to Sleipnir cores and run the rest on Senior.
Compilation

Extract actors with the following properties.

- Known buffer sizes and data types.
- Stateless actor.
- No conditional jumps.
Compilation cont’d

- Fits current toolchain of ePUMA.
- Fairly efficient code.
- Fairly common.
1 Introduction

2 ePUMA overview

3 Sleipnir

4 ePUMA toolchain

5 ePUMA programming

6 Case study - Discrete Fourier Transform (DFT)
ePUMA programming

Two simulators:
- Sleipnir simulator - kernel development
- System simulator - application development

Simulator specs:
- Simulators written in C++
- Python for scripting!
- Several prepared simulation scripts to get started fast!
ePUMA programming demo!
Case study - Discrete Fourier Transform (DFT)
Case study - Discrete Fourier Transform (DFT)

Introduction

Definition:

\[ X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} \]

Goal: Calculate it fast with FFT algorithm.

Assumptions:

- Complex data: 16 bit real part and 16 bit imaginary part.
- Twiddle factors are pre-computed offline and stored in memory.
- The DFT is computed on a single Sleipnir core.
Fast Fourier Transform

Decompose DFT into layers of butterfly-operations.
Mapping FFT to Sleipnir

Problems to solve:
- Good instruction set support
- Conflict-free parallel memory read and write
Data placement overview

Data distribution over Sleipnir local memories:

- **Input data**
- **Twiddle factors**
- **Data buffer**
- **Twiddle factors**

Typical LVM size: 4096 vectors each

Typical CM size: 128 vectors
Data access - bit reversed addressing

- FFT addressing is typically performed with simple bit-reversed addressing
- Generate addresses for 4 complex numbers/cycle
- Bank-conflicts
- Solution: Lookup table based addressing
Data access continued

Data element vectors needed for first layer:
[0, 4, 8, 12] [1, 5, 9, 13] [2, 6, 10, 14] [3, 7, 11, 15]

Simple data distribution over banks. Bank conflicts!

Bank conflicts resolved by inserting dummy elements.
Case study - Discrete Fourier Transform (DFT)

Permutation vector

Permutation vector: [0, 5, 10, 15]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>X</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
| (Actual permutation vector (since we are considering 32-bit data): [0, 1, 10, 11, 20, 21, 30, 31])

Generated address vectors:

- [0, 5, 10, 15] = 0 + [0, 5, 10, 15]
- [1, 6, 11, 16] = 1 + [0, 5, 10, 15]
- [2, 7, 12, 17] = 2 + [0, 5, 10, 15]
- [3, 8, 13, 18] = 3 + [0, 5, 10, 15]
Sleipnir permutation hardware

Constant memory is used to store permutation vectors!

- CM: Keeps permutation vectors
- Scalar AGU: Generates a scalar base address
- Vector AGU: Adds scalar base address to permutation vector to generate final addresses
Instruction set support

Butterflies can be calculated with MAC-instructions. Radix-2 butterfly:

\[
\begin{pmatrix}
  z_0 \\
  z_1
\end{pmatrix} = \begin{pmatrix}
  w_0 \\
  w_1
\end{pmatrix} \cdot \begin{pmatrix}
  1 & 1 \\
  1 & -1
\end{pmatrix} \cdot \begin{pmatrix}
  x_0 \\
  x_1
\end{pmatrix}
\]

Radix-4 butterfly:

\[
\begin{pmatrix}
  z_0 \\
  z_1 \\
  z_2 \\
  z_3
\end{pmatrix} = \begin{pmatrix}
  w_0 \\
  w_1 \\
  w_2 \\
  w_3
\end{pmatrix} \cdot \begin{pmatrix}
  1 & 1 & 1 & 1 \\
  1 & -i & -1 & i \\
  1 & -1 & 1 & -1 \\
  1 & i & -1 & -i
\end{pmatrix} \cdot \begin{pmatrix}
  x_0 \\
  x_1 \\
  x_2 \\
  x_3
\end{pmatrix}
\]
Instruction selection

Sleipnir performance:

- Complex radix-2 butterfly: 2 per cycle
- Complex radix-4 butterfly: 1 per cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cyc/Radix-4 butterfly</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC</td>
<td>4</td>
</tr>
<tr>
<td>R2 BF</td>
<td>2</td>
</tr>
<tr>
<td>R4 BF</td>
<td>1</td>
</tr>
</tbody>
</table>
Butterfly instruction example - cr4bf

Multiply with twiddle factors, followed by butterfly computation.
Implementation

System view:
- Master transfers kernel program and constant memory data to a Sleipnir core (DMA)
- Input data is transferred to the Sleipnir core
- Execute kernel computation
- Output data is transferred to main memory

Kernel work:
- Execute $x$ butterfly layers. For each layer:
  - For each butterfly:
    - Load data with input permutation
    - Multiply with twiddle factors
    - Perform butterfly
    - Store data with output permutation
Hardware organization when executing butterfly
Master software - current

```c
#include "../include/epuma_lib.h"

#define PM_START 0x1AB0
#define PM_SIZE (135*8)
#define CM_START 0x1EE8
#define CM_SIZE (28*8)
#define DIN_START 0
#define DIN_SIZE (598*8)
#define DOUT_START 0x12B0
#define DOUT_SIZE 1024*2

...
... 

```c
int main()
{
    ePUMALED(0x0000);

    ePUMASIMDSetPMBase(0, 0);
    ePUMASIMDSetCMBase(0, 0);

    // load kernel to simd and start execution
    ePUMAStarEnable(1);
    ePUMAStarLoadKernel(0, PM_START, PM_SIZE, CM_START, CM_SIZE, 1);
    ePUMAStarWait();
    ePUMAStarEnable(0);

    // load input data and coeff to simd
    ePUMAStarLoadData_1D(0, DIN_START, DIN_SIZE, 8);
    ePUMAStarEnable(1);
    ePUMAStarWait();
    ePUMAStarEnable(0);

    // store output data to main memory
    ePUMAStarStoreData_1D(0, DOUT_START, DOUT_SIZE, 8);
    ePUMAStarEnable(1);
    ePUMAStarWait();
    ePUMAStarEnable(0);

    ePUMALED(0xFFFF);
    _ePUMA_Stop();
}
```
Master software - later...

Work being done here...
.main
  out 0x10 0x230
  call LD_STAR
  wait 10*nop

  out 0x10 0x120
  call FFT

  out 0x10 0x130
  call ST_STAR
  wait 10*nop

  stop

LD_STAR:
  out 0x21 0x0 0x0
  out 0x22 VAGU_CFG 0x8000
  out 0x20 1 0
  ret

ST_STAR:
  out 0x21 0x2 0x0
  out 0x22 VAGU_CFG 256
  out 0x20 1 0
  ret
.main
dcopy ar0 c cm[ar0_init]

////////// Layer 0
dcopy ar1 c cm[ar1_init]
dcopy ar2 c cm[ar2_init]
dcopy ar1 c cm[ar1_init]
dcopy ar0 c cm[ar0_init]
dcopy vr0 m0[ar0+=8].v
1 * nop
repeat 4 1
dcopy vr1 cm[ar1+=1]
 4 * nop
 2 * cr4bf m1[ar2+=2 + cm[operm]].v m0[vr1.0 + cm[ar0+=1%]].v vr0
 2 * cr4bf m1[ar2+=2 + cm[operm]].v m0[vr1.1 + cm[ar0+=1%]].v vr0
5 * nop
...
Case study - Discrete Fourier Transform (DFT)

Sleipnir software - kernel (16-point FFT) cont.

/// Layer 1

dcopy ar1 c cm[ar1_init]
dcopy ar2 c cm[ar2_init]
dcopy vr0 m1[ar0+=8].v
dcopy vr1 m1[ar0+=8].v
dcopy vr2 m1[ar0+=8].v
dcopy vr3 m1[ar0+=8].v
1 * cr4bf m0[ar2+=2 + cm[operm]].v m1[ar1+=8 + cm[iperm0]].v vr0
1 * cr4bf m0[ar2+=2 + cm[operm]].v m1[ar1+=8 + cm[iperm1]].v vr1
1 * cr4bf m0[ar2+=2 + cm[operm]].v m1[ar1+=8 + cm[iperm2]].v vr2
1 * cr4bf m0[ar2+=2 + cm[operm]].v m1[ar1+=8 + cm[iperm3]].v vr3

10 * nop
ret
Sleipnir software - kernel (16-point FFT) cont.

.cm

// ADDRESS REGISTER SETUP VECTORS
ar0_init: coeff 0 0 0 0 0 0 0 0
ar1_init: data0 0 0 0 0 0 0 0
ca0_init: brpt brpt $(brpt+2) 0 0 0 0 0
ca1_init: brtb 0 0 0 0 0 0 0
ar2_init: data1 0 0 0 0 0 0 0

// PERMUTATION VECTORS
iperm0: 0 1 4 5 2 3 6 7
iperm1: 2 3 6 7 4 5 8 9
iperm2: 4 5 8 9 6 7 10 11
iperm3: 6 7 10 11 8 9 12 13
operm: 0 1 10 11 20 21 30 31
brpt: 0 1 10 11 20 21 30 31
brtb: 4 5 14 15 24 25 34 35
Case study - Discrete Fourier Transform (DFT)

Sleipnir software - kernel (4096-point FFT)

```
.main
dcopy ar0c cm[ar0_init]

///////// Layer 0
dcopy ar1c cm[ar1_init]
dcopy ar2c cm[ar2_init]
dcopy car1c cm[car1_init]
dcopy car0c cm[car0_init]
dcopy vr0 m0[ar0+=8].v
1 * nop
repeat 10 16
  dcopy vr1 cm[car1+=1]
  4 * nop
  8 * cr4bf m1[ar2+=2 + cm[operm]].v m0[vr1.0 + cm[car0+=1%]].v vr0
  8 * cr4bf m1[ar2+=2 + cm[operm]].v m0[vr1.1 + cm[car0+=1%]].v vr0
  8 * cr4bf m1[ar2+=2 + cm[operm]].v m0[vr1.2 + cm[car0+=1%]].v vr0
  8 * cr4bf m1[ar2+=2 + cm[operm]].v m0[vr1.3 + cm[car0+=1%]].v vr0
  8 * cr4bf m1[ar2+=2 + cm[operm]].v m0[vr1.4 + cm[car0+=1%]].v vr0
  8 * cr4bf m1[ar2+=2 + cm[operm]].v m0[vr1.5 + cm[car0+=1%]].v vr0
  8 * cr4bf m1[ar2+=2 + cm[operm]].v m0[vr1.6 + cm[car0+=1%]].v vr0
  8 * cr4bf m1[ar2+=2 + cm[operm]].v m0[vr1.7 + cm[car0+=1%]].v vr0
7 * nop
...
```
Case study - Discrete Fourier Transform (DFT)

Sleipnir software - kernel (4096-point FFT) cont.

```
... 

///////// Layer 1
dcopy ar1c cm [ ar1_init ]
dcopy ar2c cm [ ar2_init ]
dcopy vr0 m1 [ ar0 +=8 ] . v
dcopy vr1 m1 [ ar0 +=8 ] . v
dcopy vr2 m1 [ ar0 +=8 ] . v
dcopy vr3 m1 [ ar0 +=8 ] . v
256 * cr4bf m0 [ ar2 +=2 + cm [ operm ] ] . v m1 [ ar1 +=8 + cm [ iperm0 ] ] . v vr0
256 * cr4bf m0 [ ar2 +=2 + cm [ operm ] ] . v m1 [ ar1 +=8 + cm [ iperm1 ] ] . v vr1
256 * cr4bf m0 [ ar2 +=2 + cm [ operm ] ] . v m1 [ ar1 +=8 + cm [ iperm2 ] ] . v vr2
256 * cr4bf m0 [ ar2 +=2 + cm [ operm ] ] . v m1 [ ar1 +=8 + cm [ iperm3 ] ] . v vr3
7 * nop

... 
```
## DFT benchmarks

Cycle times on 1 Sleipnir core (incl. prologue and epilogue):

<table>
<thead>
<tr>
<th># points</th>
<th>DFT</th>
<th>FFT w. perm.</th>
<th>FFT w. R2BF</th>
<th>FFT w. R4BF</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>81</td>
<td>71</td>
<td>55</td>
<td>47</td>
</tr>
<tr>
<td>64</td>
<td>1053</td>
<td>260</td>
<td>164</td>
<td>116</td>
</tr>
<tr>
<td>256</td>
<td>16461</td>
<td>1184</td>
<td>672</td>
<td>416</td>
</tr>
<tr>
<td>1024</td>
<td>262413</td>
<td>5569</td>
<td>3009</td>
<td>1729</td>
</tr>
<tr>
<td>4096</td>
<td>4195341</td>
<td>26102</td>
<td>13814</td>
<td>7670</td>
</tr>
</tbody>
</table>
ePUMA FFT demo

Demo!
End