ADAPTEVA FOR BASE STATION SIGNAL PROCESSING

Benchmarking the worlds most FLOPS/W-efficient computer

Martin Lundqvist, David Engdal & Peter Brauer at Ericsson AB
OUTLINE

› Background, Why Benchmarking? [David, 5’]
› Adapteva: [David, 15’]
  – The company and its product
  – Some raw numbers
› Our application: Baseband signal processing [David, 10’]
› The platform, “BOS”, and our application on the Epiphany [Martin, 30’]
› Conclusions [Martin + David, 10’]
WHY BENCHMARKING?

The Antikythera mechanism, 
~90 B.C.

The Zuse Z3, 1941.

Gradual but sometimes disruptive development
WHY BENCHMARKING?

CPU:

Very hetero-

geneous SoC:

Gradual but sometimes disruptive development
WHY BENCHMARKING?

› Warning! Do not engage yourself in these shady things!
  – Embedded systems (SW+HW) are always tailored to their application
  – You will mostly be criticized for being unjust or foolish by anyone who see your results. And probably for good reasons…

› The only thing more foolish than doing benchmarking of embedded systems is not to do it.

Caveat: Everything I say from here on is only part of the truth
WHY BENCHMARKING?

› Our mission is to look at:
  – SoCs similar to and competing with ours
  – New ways to solve the need for compute power

Gradual but sometimes disruptive development
An Introduction to the Epiphany Manycore Architecture (Andreas Olofsson, Halmstad-2012)
– Cherry-picking from pages 1-35 (depending on interest and time available)
SOME RAW NUMBERS

› Clock cycle count tested by us.
› Other data from data sheets, websites and papers.

› We have compared the Epiphany to the Xeon X5680 as we did have some experience of that one.
› For comparison, numbers based on data sheet calculations for the Intel Xeon Phi 5110P are also provided.

› Remember, this is only about raw DSP power.
1D-FFT: EPIPHANY VS INTEL XEON X5680

› Complex ffts (single precision):
  - Xeon X5680: 2048 pts, 38000 cc => (3.3 GHz) 11.5 µs
  - Epiphany: ~100 cc/pts => 200000 => (650 MHz) 308 µs
  - Xeon X5680, FFT per energy:
    › One of six cores => 140/6=23 W power.
    › 11.5 µs * 23 W = 265 µJ per 2048 complex FFT
  - Epiphany, FFT per energy:
    › One of 64 cores => 2/64=31 mW power.
    › 308 µs * 31 mW = 9.6 µJ per 2048 complex FFT

› Epiphany 27 times slower but 27 times more energy efficient.
1D-FFT: EPIPHANY VS INTEL XEON X5680

› Size:
  – Xeon X5680: 41 mm² per core (32nm) ➞ 2100 FFTs/s per mm²
  – Epiphany: 0.1 mm² per core (28nm) ➞ 32500 FFTs/s per mm²

› ➞ Epiphany is 15 times as area efficient
1D-FFT: EPIPHERANY VS INTEL XEON PHI

› Energy:
  – Xeon Phi 5110P, TDP = 225 W, FLOP Performance ~7 times X5680
  – Epiphany 4 times more energy efficient.

› Size:
  – Xeon Phi 5110P (22nm): \(\frac{350}{60} = 5.8 \text{ mm}^2\)
  – FFTs per Size: 10500 FFTs/s per mm\(^2\)
  – Epiphany is 3 times as area efficient
OUR APPLICATION
BASEBAND?

Radio

Common processing

Per user processing

Transport
**PREREQUISITES**

- Large SW projects, >> 50 SW designers
- > 200,000 lines of soft real time C-code
- Reuse of legacy code
- Time to market – do it right the first time
- SW Quality – ability to test at several levels (modul…)
- Model based functional systemization
- Hardware cost and energy consumption still seen as very important
• The major part of the baseband works in a time structure set by the air frames (of about 1 ms duration).

• The data scheduler has close to perfect knowledge about what will happen in the next air frame.

• The control part of the baseband works on a longer time scale and with less strict real time requirements.

One straightforward approach could be to statically allocate the control tasks to a few cores.

The user data tasks are allocated on a frame-by-frame basis using the knowledge from the data scheduler.
SIMPLIFIED UL-PROBLEM

The UL part is normally the most complex, so let’s focus on that one for the moment. Here the dynamics in problem are illustrated.
THE DYNAMIC CHALLENGE
THE DYNAMIC CHALLENGE — TO MAKE IT EVEN WORSE...

One big user together with 50 small users

“Algorithm split”

Latency limit

“Embarrassingly parallel”

#simultaneous users
THE DYNAMIC CHALLENGE – TA MAKE IT EVEN WORSE...

Three different configurations needs to be processed at the same time
It’s a dynamic bunch of graphs!
“BOS”, AND OUR APPLICATION ON EPIPHANY
SAME APPLICATION – DIFFERENT CONFIGURATIONS

CONFIG 1

CONFIG 2

$t_A$

$t_B$

$t_C$

$L_1$

$L_2$
LATENCY IMPROVEMENTS

\[ L'_A > L_A/3 \]
PARALLEL PUNISHMENT

Latency – serial case
\[ L_s = T_{s1} + T_p + T_{s2} \]

Latency – parallel case
\[ L_p = T_{s1} + T_{sf} + T_{pf} + T_p/N + T_{pj} + T_{sj} + T_{s2} \]

Latency gain
\[ L_s - L_p = T_p - (T_{sf} + T_{pf} + T_p/N + T_{pj} + T_{sj}) \]
Lateness improvements vs resource consumption
LATTENCY IMPROVEMENTS VS RESOURCE CONSUMPTION
LATENCY IMPROVEMENTS
VS RESOURCE CONSUMPTION

Latency improvements vs resource consumption

Adapteva for baseband? | © Ericsson AB 2013 | 2013-09-10 | Page 27
ADAPTEVA PREREQUISIT ES

• No real-time platform

• Limited local memory

• No external memory

• No real-time tracing

...det ger sig!
GRAPH ANATOMY

**Task** – Representation of a kernel’s flow dependency

**Kernel** – Encapsulated function with well defined interfaces, runs until completion on one core.

**Graph** – Complete, abstract flow of tasks and barriers

**Barrier** – Representation of inter-task dependency and synchronization

**Core** – Resource in a multicore system.
DATA MOVEMENT - STRUCTURE / BEHAVIOR

config_s{
    static_var_1;
    static_var_2;
    ...
    dynamic_var_1;
    dynamic_var_2;
    ...
}

k2_indata_s{
    config;
    var_i2a;
    var_i2b;
}

k4_indata_s{
    config;
    var_i4a;
}

k2_outdata_s{
    var_o2a;
}

k4_outdata_s{
    var_o4a;
    var_o4b;
}

move_data_2_6( )
move_data_2_8( )
move_data_4_6( )
move_data_4_8( )

k6_indata_s{
    config;
    var_i6a;
    var_i6b;
    var_i6c;
}

k8_indata_s{
    config;
    var_i8a;
    var_i8b;
}

move_data_2_6( )
move_data_2_8( )
move_data_4_6( )
move_data_4_8( )
First of t1, t2 to reach b1:

- responsible to find cores supporting k6 and k8
- copies the static config part
- initiates the dynamic config part

Everyone of t1, t2:

- copies kernel data

Every kernel transition has its specific logic, when copying data between instances of different dimensions.
PRACTICAL PARALLELIZATION

single resource (maximal serialism)

logical view
PRACTICAL PARALLELIZATION

single resource (maximal serialism)

logical view

infinite resources (maximal parallelism)
PRACTICAL PARALLELIZATION

single resource (maximal serialism)

finite resources (limited parallelism)

infinite resources (maximal parallelism)
CONCLUSIONS

› What it comes with:
  – Top performance in general purpose and DSP FLOPS/W
  – OpenCL: for batch oriented computations (with large number of FLOPS per IO data bit)
  – Well balanced local memory vs compute performance (for our domain)
  – Easy programming (of cores) in plain C
  – Very scalable non-SIMD-hardware promises easy reusable software.

› What it lacks:
  – Cache memory (Are you willing to program without caches?)
  – Platform SW for efficient execution of task graphs
  – Some investments in SW the department, e.g. on trace & debug

› Who is it up against?
  – Yes: Intel Xeon Phi, GP-GPU, ARM (also with Neon), DSPs
  – No: Base station SoCs (e.g. Qualcomm FSM99xx)

Epiphany can be a good building block for SoCs
REFERENCES

› [2] A Sub 2 Watt 64-core 100 GFLOPS Accelerator Programmable in C/C++ or OpenCL (HPEC-2012)
› [4] Intel for Baseband Benchmark (Ericsson internal study, 2011)
BACKUP SLIDES
'Whiteboard' design

XML format

```
<task>
  <name>t1</name>
  <kernel>kernel_a</kernel>
  <barrier>b1</barrier>
</task>

<task>
  <name>t2</name>
  <kernel>kernel_b</kernel>
  <barrier>b3</barrier>
</task>

<task>
  <name>t3</name>
  <kernel>kernel_c</kernel>
  <barrier>b2</barrier>
</task>

<task>
  <name>t4</name>
  <kernel>kernel_d</kernel>
  <barrier>b3</barrier>
</task>

<task>
  <name>t5</name>
  <kernel>kernel_a</kernel>
</task>
```

Host info - dynamic

```
task_id 123:
  task_id: t1
  kernel_id: 8
  parallel_task_id: 0
  barrier_id: 99

task_id 234:
  task_id: t2
  kernel_id: 33
  parallel_task_id: 0
  barrier_id: 24

...```

Host info - static

```
kernel_id 1:abcdefgh
kernel_id 2:qwerty
kernel_id 3:x_y_z
...```

Interface format

```
task_id=123
kernel_id=8
parallel_task_id=0
barrier_id=99

...```

How is the host to keep track of the running task_id's and barrier_id's for the current session?

• One translation file per ongoing session – dynamically adding and deleting rows as tasks and barriers are created and consumed?

```c
typedef struct{
  char name[];
  char kernel[];
  char barrier[];
  host_task_t;
} host_task_t;
```

```c
typedef struct{
  unsigned short id;
  unsigned short kernel_id;
  unsigned short parallel_task_id;
  unsigned short barrier_id;
} if_task_t; // 8 bytes
```

```c
typedef struct{
  unsigned short id;
  unsigned short semaphore;
  unsigned short task_id;
} if_barrier_t; // 6 bytes
```

typedef struct{
  char name[];
  char task[][];
} host_barrier_t;
Graph sanity in interface format:

- Only one first task
- Only one last task
- First task first
- Parallel task 'after' pointing task

```
target_id=123
kernel_id=8
parallel_task_id=0

barrier_id=33
task_id=234
kernel_id=33
parallel_task_id=345

barrier_id=24
task_id=345
kernel_id=1
parallel_task_id=0

barrier_id=24
task_id=456
kernel_id=63
parallel_task_id=0

barrier_id=99
kernel_id=63
parallel_task_id=0

barrier_id=99
kernel_id=1
parallel_task_id=0

barrier_id=24
kernel_id=8
parallel_task_id=0

barrier_id=24
kernel_id=8
parallel_task_id=0

semaphore=1

barrier_id=99
kernel_id=8
parallel_task_id=0

barrier_id=7
semaphore=1

barrier_id=24
semaphore=2

barrier_id=99
semaphore=1
task_p

barrier_id=7
semaphore=1
task_p

barrier_id=24
semaphore=2
task_p

typedef struct {
  unsigned short id;
  unsigned short kernel_id;
  task_t *parallel_task;
  barrier_t *barrier_p;
} task_t; // 12 bytes

typedef struct {
  unsigned short id;
  int lock;
  unsigned short:15 semaphore;
  unsigned short:1 reached;
  task_t *task;
} barrier_t; // 12 bytes
```
GRAPH SANITY

- There shall be only one first task
- There shall be only one last task
MEMORY STRUCTURES
How large is the:

- BOS code section?
- Kernel function code section? Each kernel code?
- Trace buffer?
- Heap? Stack?

So, how large can the graph area be?
A posted interface graph consists of max T tasks of 8 bytes, and B barriers of 6 bytes. (Totaling X bytes for 20 tasks and 10 barriers.)

An activated graph consists of max T tasks of 12 bytes, and B barriers of 12 bytes. (Totaling Y bytes for 20 tasks and 10 barriers. Five simultaneous graphs totals Z bytes, including first task pointers.)

Decrease one core’s kernel code accordingly.

The host area is posted to by host, when possible, and activated by target, after discovery. No lock is necessary.

Task, barriers, and first task pointers are added by core 1, and read and deleted by any core.
typedef struct {
    ushort write_offset; // next byte to write
    ushort read_offset;  // next byte to read if not equal to write_p
    char congestion;     //
    char buffer[SIZE];   // buffer of bytes
} trace_buffer_t;

- Trace area is a byte-size based buffer.
- At startup, read_offset and write_offset 'points' to first byte.
- When empty, read_offset and write_offset points to same byte.
- Writing when empty re-initiates write_offset and read_offset to first byte.
- When overfilled, congestion is set, to denote missing trace info.
- No locks or mutexes!

Trace format suggestion:
- HEADER
  - length
  - type
  - time stamp
- DATA
- Copy interface-t1 to target-t1 (incomplete task and barrier pointers)
  - Interface-t1 now obsolete
- Let unfinished_task[0] -> target-t1
...
- Copy interface-t5 to target-t5 (incomplete task and barrier pointers)
  - Interface-t5 now obsolete

- Find parallel task to unfinished_task[0..4] (t1->NULL, t2->t3, ... , t3,t4,t5->NULL)
  - Complete unfinished tasks’ parallel task pointer

- Copy interface-b1 to target-b1 (incomplete task pointer)
- Find unfinished tasks 'pointing' to current barrier’s id (t1->b1)
  - Complete unfinished tasks' barrier pointer
- Finish unfinished task (unfinished_task[0])
- Find task that current barrier points to (b1->t2)
  - Complete current barrier’s task pointer / finish current barrier
...
- Copy interface-b3 to target-b3 (incomplete task pointer)
- Find unfinished tasks 'pointing' to current barrier’s id (t2->b3, t4->b3)
  - Complete unfinished tasks’ barrier pointer
• t1 finishes while t2 is running
  • t1 locks b1 for exclusive use
    • t1 keeps lock until at least one of t3 and t4 gets assigned to a core
  • b1 not previously reached – reservation responsibility
    • checks self, then all cores once, for t3 / t4 kernel support
  • **self / other**
    • Sets core’s state to BUSYRESERVED / RESERVED
    • Sets core’s next_task_id / task_id
    • Copies t1’s outdata to reserved core’s indata
  • t1 decreases b1’s semaphore and unlocks b1
    • t1 releases lock on b1 when at least one task assigned
    • t1 keeps checking for kernel support until every task assigned
  • t1 sets its own state to IDLE / BUSY
    • t1 waits x us to be reserved
    • if x us passes, t1 checks for new graph (‘first task pointer’)
    • if no new graph found, t1 enters sleep mode
t2 is running

- t2 finishes last
  - t2 locks b2 for exclusive use (gets lock in this scenario)
    - t1 keeps lock until at least one of t3 and t4 gets assigned to a core
  - t2 checks if t3 or t4 is assigned (it is in this scenario)
    - Meaning t2 is not the first task to reach b1
    - t2 copies its outdata to reserved core’s indata
  - t2 decreases b1’s semaphore and unlocks and discards b1
  - t2 sets its own state to idle
    - t2 waits x us to be reserved
    - if x us passes, t2 checks for new graph (‘first task pointer’)
    - if no new graph found, t2 enters sleep mode
DATA MOVEMENT – STRUCTURES / BEHAVIOR II
If \( t_1 \) finishes first, and it’s considered optimal to reserve the same core to run \( t_3 \), shouldn’t it be considered optimal to reserve the core running \( t_2 \) to run \( t_4 \), too? (Not possible with current solution for graph description.)

We could need a run-time scheduler that parallelizes any task, ’too big’ to fit in the total latency budget. Based on static info, concerning the actual kernel, and dynamic info based on certain parameters.