Low-level Parallel Programming: Explicit DMA Transfers on the Cell

Contents of Lecture 7

- An Introduction to the Cell Broadband Engine Architecture

Key Ideas of the Cell Processor

- Highest possible performance at a low power dissipation
- 8 SIMD vector processors added to a multithreaded PowerPC processor
- Cost: more complex programming
- Benefit for the right program: superior performance per watt.

History of the Cell Processor

1999 A partnership with IBM is proposed by Sony CEO Nobuyuki Idei.

2001 Sony, Toshiba and IBM form the STI Design Center in Austin.

2004 First Cell processor manufactured and runs faster than 4 GHz.

2005 PS3 is shown by Sony.

2006 PS3 is sold by Sony and IBM releases a Linux SDK for the PS3.

2008 IBM's Roadrunner with 12,960 Cell processors and Opterons becomes the fastest computer in the world.

2009 There were (probably false) rumours that the Cell was killed.

2010 IBM say they might integrate Cell technology in normal Power processors.

The Cell Broadband Engine Architecture

- A Cell processor consists of:
  - One multithreaded 64-bit PowerPC processor (currently two threads)
  - Eight SIMD vector SPU processors with 128 registers and 256 KB RAM
  - One programmable memory interface controller
  - Two input/output interfaces
- The PowerPC processor is optimized for general purpose computing and SIMD vector processing
- The SPUs (synergistic processor units) are optimized for SIMD vector processing and concurrent data transfers.
- The SPUs have no caches and both data and instructions must be explicitly transferred from system memory by software using DMA.
- If programmed cleverly, the SPUs can achieve **very high** performance.

The Synergistic Processor Unit

- The SPU contains two pipelines, the even and the odd pipelines.
  - Even: fixed point unit and floating point unit.
  - Odd: fixed point unit, load/store unit, and Channel and DMA unit.
- 128 128-bit registers.
- All ISO C integer data types and float and double.
- 256 KB local RAM memory for instructions and data.
- No hardware caches!
- The Channel and DMA unit is used for synchronization with other processors and to transfer data.

Filling the SPU Instruction Pipeline

- Firstly, the instruction must already be in the local store.
- Before an instruction can start doing work in a functional unit in one of the even or odd pipelines, it must go through the following:
  1. Fetch: 7 cycles.
  2. Decode: 3 cycles.
  3. Issue: 2 cycles.
- Instructions are fetched into one of ILB1 and ILB2, which can hold 32 instructions each.
- When an ILB is empty 32 new instructions are fetched from the local store. Every instruction is 32 bits.
- Instruction fetch has the lowest priority after DMA and load/store accesses.

Branch Prediction in the SPU

- There is no dynamic branch prediction in the SPU.
- Branches to lower addresses are predicted to be taken. The idea is that a loop's branch to the next iteration is backward and is usually taken.
- A mispredicted branch costs 18 cycles.
- There are also special **branch hint** instructions which software can use to tell SPU about future branches.

SIMD Programming on the Cell

- Both the PPU and SPU support SIMD instructions.
- On the PPU, it's the VMX or AltiVec instruction set.
- On the SPU, all instructions are SIMD.
- The two SIMD instruction sets are similar but not equal.
- Use the vector extensions in GCC to avoid having to use specific SIMD instructions.

EIB and MFC

- EIB stands for the **Element Interconnect Bus**
- MFC stands for the **Memory Flow Controller**
- These two perform the data transport for DMA transfers.
- There is one EIB which consists of four rings, each 16 bytes wide.
- Each SPU has its own MFC which send and receive messages on the EIB.
- Two rings send data in one direction and the other two in the other direction.
- Software doesn't specify which ring should be used — too messy and which is best to use is dependent on what is currently happening.
- Each ring can have up to three concurrent transfers — but they are not allowed to overlap, e.g. SPU 2 can send to SPU 5 and SPU 6 to SPU 7 concurrently but SPU 6 cannot concurrently send to SPU 4 on the same ring — but on another ring is OK.

DMA Transfers

- The size of a DMA transfer must be 1,2,4,8, or a multiple of 16 and at most 16 KB.
- Better performance for 128-byte aligned data.
- The start address of objects must be 16-byte aligned — allocate extra memory with malloc.
- The lower 4 bits of the effective address and the LS address must be equal.
- If above rules are not followed, the program will get a bus error and terminate.

DMA Parameters and Operation

- A DMA request needs the following parameters:
  - effective address
  - local store address
  - size
  - tag
  - operation
- The tag identifies the DMA request and is a value in the range 0..31.
- The operation is either `MFC_PUT_CMD` or `MFC_GET_CMD`.
- This looks like a normal memory load or store access, except for the tag and larger data size.
- What is so special that the Cell processor?

Comparison with Data Prefetching

- Recall that cache misses can be difficult to avoid on multiprocessors and that data prefetching is not a universal solution.
- For instance the prefetched data might be evicted before being used — even by other prefetches!
- The difference between normal multicores (multiprocessors) and the Cell is that you have complete control over the local store parameter in the DMA request.
- You can avoid overwriting useful data simply by using different local store addresses.
- More control in other words.
- The DMA operations are similar to data prefetches in that the processor continues with other work and the MFC performs the data transfer.

The Local Store is Not a Cache, Though!

- If prefetched data does not arrive in the cache before it's needed, the processor will wait until it does arrive.
- That will not happen automatically in the SPU.
- Recall that the tag DMA parameter is used to identify DMA request.
- You can tell the SPU to wait until a subset of all pending DMA operations have completed.
- This is done by setting a bit corresponding to the tag you want to wait for and then issue a special command.
- This will cause the SPU to wait until all DMA operations of the specified tags have completed.
- In fact, you can use one tag for multiple DMA operations so you can have more than 32 pending operations if you wish.
- With this architecture you control parallelism between SPU's as well as data transfer and computation within each SPU.

Mailboxes

- An alternative to DMA for small amounts of data is using mailboxes.
- Each SPU has three mailboxes:
  - An outgoing mailbox
  - An outgoing interrupt mailbox
  - An incoming mailbox
- Directions are from the view of an SPU:
  - out means out from the SPU, and
  - in means in to the SPU.
- The outgoing mailbox has a capacity of one 32-bit word.
- The incoming mailbox has a capacity of four 32-bit words.
- Writing to a full or reading from an empty mailbox blocks the thread.

Software Development for the Cell

- There are several Linux distributions ported to the Cell and we will use YDL.
- YDL = Yellow Dog Linux. The Y is the Y in the command `yum` (Yellowdog Update Manager) available at least on YDL and on Fedora.
- IBM distributes the Cell Software Development Kit for YDL.
- The SDK contains numerous libraries, documention, tutorials, example programs.
- A Cell application is written in multiple parts:
  - One part for the PPU as a normal C program.
  - One part for each program the SPU's should execute.
  - All of these have their own `main` function.
  - The different parts are compiled with one of:
    - `spu-gcc`, and
    - `ppu32-gcc`.

Connecting the PPU and SPU Programs

- Suppose your SPU executable is called `dataflow`.
- All executables for the SPU are statically linked.
- The SPU executable is made accessible to the PPU program through:
  1. `ppu-embedspu -m32 dataflow dataflow dataflow-embed.o`
     The first `dataflow` is the symbol name and the second is the input file name.
  2. Declare in the PPU program:
     `extern spe_program_handle_t dataflow;`
     This external symbol is loaded by the PPU program and then the SPU can be executed.
- The PPU thread which starts the SPU program is blocked until the SPU program exits.
- With Pthreads, we use a thread to run the SPU and wait for it to complete.

Executing an SPU Program from a PPU Program

```
extern spe_program_handle_t     dataflow;
spe_context_ptr_t               ctx;
unsigned int                    entry = SPE_DEFAULT_ENTRY;

ctx = spe_context_create (0, NULL);
spe_program_load (ctx, &dataflow);
spe_context_run(ctx, &entry, 0, arg, NULL, NULL);
```

- Create one context for each SPU.
- Call `spe_context_run` from a thread on the PPU.
- The `void* arg` argument becomes accessible to the SPU program through an `unsigned long long parm`:

```
/* SPU main. */
int main(unsigned long long id, unsigned long long parm);
```

SPU makefile

```
PROGRAM_spu     := dataflow
LIBRARY_embed   := lib_dataflow_spu.a

INCLUDE         := -I ..  -I /opt/cell/sdk/src/lib/sync/spu_source

LDFLAGS         += -lsync -L/opt/cell/sdk/src/lib/sync/spu
CFLAGS          += -O4

include /opt/cell/sdk/buildutils/make.footer
```

PPU makefile

```
C               = 3
S               = 10000
V               = 1000
U               = 4
A               = 1000

DIRS            := spu
PROGRAM_ppu     := dataflow
IMPORTS         = spu/lib_dataflow_spu.a -lspe2 -lpthread

INSTALL_DIR     = $(EXP_SDKBIN)/tutorial
INSTALL_DIR     = ..
INSTALL_FILES   = $(PROGRAM_ppu)

include /opt/cell/sdk/buildutils/make.footer

#CFLAGS         = -O4 -Wall -pedantic -std=c99 -DTEST
#CFLAGS         = -O4 -Wall -pedantic -std=c99
CFLAGS          = -maltivec -O4 -Wall -pedantic -std=c99 -Werror

OBJS    = driver.o error.o

all: $(OBJS)
        ./dataflow $(S) $(V) $(U) $(A) $(C)

clean:
        rm -f $(PROGRAM_ppu) $(OBJS)
        (cd spu; make clean)
```

Hints for Cell Dataflow Analysis

- Be careful to align data properly otherwise you will see bus errors.
- Change the data structures from the C program to fit the Cell better.
- Parallelize on three levels:
  1. Compute using multiple SPU's.
  2. Transfer data in both directions while computing on an SPU.
  3. Use `__vector` GCC extension to exploit SIMD on the SPU's.
- Premature optimization is the root of all evil.

More About Tags

- Recall, every DMA request uses a tag 0..31 to identify the **tag group** of the request.
- A new tag is allocated with:

```
tag = mfc_tag_reserve();
```

- Alternatively, additional DMA requests can be grouped to a previously allocated tag.
- Nothing stops you from using only one tag for your entire program but it will not be very efficient.
- Recall, the purpose of using different tags is to have multiple data transfers in progress concurrently with your computations.
- E.g. one tag group for iteration $i + 2$, one for $i + 1$, one for i which you must wait for, and then other groups for writing back data to system RAM.

Waiting for Tag Group Completion

- `mfc_read_tag_status_immediate` returns an int with bits set to one for the completed tag groups.

```
/* loop until tag 3 has completed: */
do {
        work();
        status = mfc_read_tag_status_immediate();
} while ((status & (1<<3)) == 0);
```

- With this function, we can actually do some more work while waiting...
- The next two functions block the SPU until some or all tag groups have completed.
- `mfc_read_tag_status_any` returns when one has completed.
- `mfc_read_tag_status_all` returns when all have completed.

Releasing a Tag

- A tag is released using the function
  `uint32_t mfc_tag_release(uint32_t tag);`
- The functions for reserving and releasing a tag are not necessary to use.
- We can, however, actually use any value in 0..31.
- If different parts of our application need a tag, these library functions can help us not using the same tag for different purposes at the same time.