Programming the Cell BE

Second Winter School
Geilo, Norway

André Rigland Brodtkorb
<Andre.Brodtkorb@sintef.no>

SINTEF ICT
Department of Applied Mathematics

2007-01-25

Outline

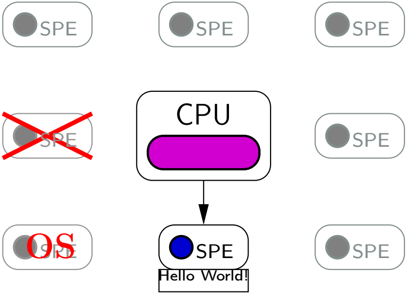
1. Briefly about compilation.
2. Hello World:
   - using *one* SPE,
   - using *all* SPEs,
   - using *all* SPEs *and* Direct Memory Access (DMA).
3. Discrete Cosine Transform (JPEG).
4. Summary

- Workshop-like presentation with real code:)
- You can download the code from `http://babrodtk.at.ifi.uio.no/Research/Cell` now!

Compilation

- You need a PS3, a Cell blade server, a Mercury Cell card, or a Cell simulator.
- You need a special compiler (IBM XL C/C++ or modified GCC).
- SPE code is embedded into PPE code.
- Embedded SPE code is linked together with the PPE code to one executable.
- Keep it in your mind that the compiler can contain bugs.

Hello World



1. Run PPE program
2. PPE program starts the SPE
3. The SPE writes "Hello World"

Hello World on one SPE.

- We need a datastructure to store SPE program information:
  - Entry point (address).
  - Arguments.
  - ID of SPE context.
- We need code that will execute on the SPE.
- We need to create the contexts, and start them on the PPE.

HelloWorld.cpp (PPE)

```
typedef struct {
  spe_context_ptr_t speid; //< Identifier for SPE context
  void* argp;              //< Pointer to arguments
  void* envp;              //< Pointer to environment
  unsigned int entry;      //< Entry point of the spe
} helloWorldContext;

extern spe_program_handle_t HelloWorld_spe;

int main(int argc, char** argv) {
  spe_stop_info_t stop_info;
  helloWorldContext context;
  int status;
  unsigned int flags = SPE_EVENTS_ENABLE;

  context.speid = spe_context_create(flags, NULL);
  context.argp = NULL; //Not used
  context.envp = NULL; //Not used
  context.entry = SPE_DEFAULT_ENTRY;

  status = spe_program_load(context.speid, &HelloWorld_spe);
  status = spe_context_run(context.speid, &context.entry,
                           0, context.argp, context.envp, &stop_info);
  spe_context_destroy(context.speid);
}
```

Running the SPE program

HelloWorld.cpp - excerpt (PPE)

```
...
status = spe_context_run(context.speid, &context.entry,
                         0, context.argp, context.envp,
                         &stop_info);
...
```

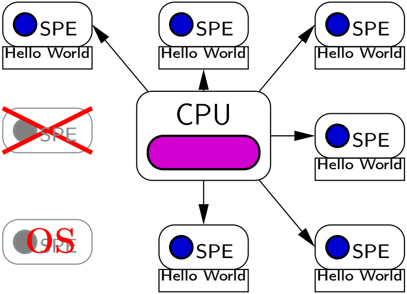
HelloWorld_spe.cpp (SPE)

```
int main(unsigned long long speid,
         unsigned long long argp,
         unsigned long long envp) {
  printf("Hello world!\n");
  return 0;
}
```

Summary one SPE

- We have a struct that keeps track of the SPE context.
- The PPE creates the SPE context, and loads the program into that context.
- The PPE runs the context, and waits for it to complete.

Hello world from All



1. Run PPE program
2. PPE program starts all SPEs using pthreads
3. All SPEs write "Hello World"

Hello World using all SPEs

- Find out how many SPEs are available.
- Use pthreads on the PPE to execute different SPE contexts.
- Wait for the SPE context to complete, and thus the PPE thread to join.

HelloWorldFromAll.cpp - helloWorldContext (PPE)

```
typedef struct {
  spe_context_ptr_t speid; //< Identifier for SPE context
  pthread_t threadid;      //< Identifier for PPE thread
  void* argp;              //< Pointer to arguments sent to the spe
  void* envp;              //< Pointer to environment sent to the spe
  unsigned int entry;      //< Entry point of the spe
} helloWorldContext;
```

HelloWorldFromAll.cpp - main (PPE)

```
int main(int argc, char** argv) {
  unsigned int flags = SPE_EVENTS_ENABLE;
  int status;

  int n = spe_cpu_info_get(SPE_COUNT_PHYSICAL_SPES, -1);

  helloWorldContext* contexts = new helloWorldContext[n];

  for(int i=0; i<n; ++i) { //Create context and load program
    contexts[i].speid = spe_context_create(flags, NULL);
    contexts[i].argp = NULL; //Not used
    contexts[i].envp = NULL; //Not used
    contexts[i].entry = SPE_DEFAULT_ENTRY;

    status = spe_program_load(contexts[i].speid, &HelloWorld_spe);
  }
  for(int i=0; i<n; ++i) { //Run the contexts
    status = pthread_create(&contexts[i].threadid, NULL,
                            &create_spe_thread, &contexts[i]);
  }
  for(int i=0; i<n; ++i) { //Wait for pthreads to join
    status = pthread_join(contexts[i].threadid, NULL);
  }

  delete [] contexts;
}
```

Pthread function

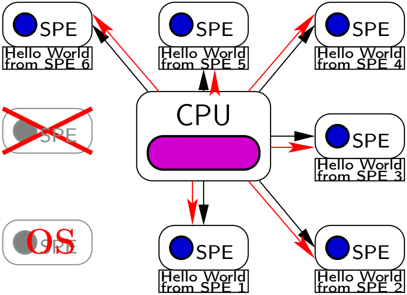
HelloWorldFromAll.cpp - create_spe_thread (PPE)

```
void* create_spe_thread(void* context_) {
  spe_stop_info_t stop_info;
  helloWorldContext* context;
  int status;

  context = (helloWorldContext*) context_;

  status = spe_context_run(context->speid, &context->entry,
                           0, context->argp, context->envp,
                           &stop_info);
  spe_context_destroy(context->speid);

  pthread_exit(NULL);
}
```

Summary all SPEs

1. The PPE creates the contexts, and loads the SPE programs.
2. Then, the PPE creates one pthread per physical SPE.
3. The pthread executes the `create_spe_thread` function, which runs the SPE context.
4. The PPE thread exits when the SPE context has completed.

Hello world with Direct Memory Access (DMA)



1. Run PPE program
2. PPE program starts all SPEs
3. All SPEs DMA an integer over to local store
4. All SPEs write “Hello World from SPE %i”

Hello World with DMA.

- DMA - Direct Memory Access.
- Explicit data movement, or “software cache”.
- The SPEs (16 long queue per SPE) are more efficient than the PPE (8 long queue).
- 16 byte alignment required (128 preferable).
- DMA size must be 1, 2, 4, 8 or $n \times 16$ byte (multiple of 128 preferable).
- Maximum DMA size is 16 Kbyte
- “Assembly” (SPE intrinsics).

Allocating aligned data

HelloWorldDMA.cpp - excerpt (PPE)

```
int main(int argc, char** argv) {
  ...
  for(int i=0; i<n; ++i) { //Create context and load program
    contexts[i].speid = spe_context_create(flags, NULL);
    contexts[i].argp = malloc_align(sizeof(int), 16);
    contexts[i].envp = NULL; //Not used
    contexts[i].entry = SPE_DEFAULT_ENTRY;

    int* spe_argp = (int*) contexts[i].argp;
    *spe_argp = i;

    status = spe_program_load(contexts[i].speid,
                              &HelloWorldDMA_spe);
  }
  ...
}
```

Direct Memory Access

HelloWorldDMA_spe.cpp (SPE)

```
int main(unsigned long long speid,
         unsigned long long argp,
         unsigned long long envp) {
  int i __attribute__((aligned(16)));

  spu_writech(MFC_WrTagMask, -1);

  void* lsa = (void*) &i;
  unsigned int ea = argp;
  unsigned int size = sizeof(int);
  unsigned int tagid = 0;
  unsigned int cmd = MFC_GET_CMD;

  spu_mfcdma32(lsa, ea, size, tagid, cmd);
  spu_mfcstat(MFC_TAG_UPDATE_ALL);

  printf("Hello world from SPE %d!\n", i);

  return 0;
}
```

Summary DMA

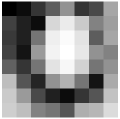
- DMA on the Cell requires aligned memory!
- DMA does not stall the processor.
- The SPEs can queue more DMA commands than the PPE.
- Compute whilst waiting for data.

MJPEG compression using the Cell BE

Algorithm

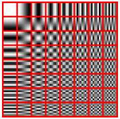
1. Divide the input image into 8×8 blocks
2. For each block
   1. **Convert from discrete pixels to discrete cosines (DCT).**
   2. **Quantize (element division by a fixed 8×8 matrix).**
   3. Run through the result matrix in a "zigzag" pattern, and store nonzero coefficients.
   4. Huffman code the vector.

(M)JPEG algorithm



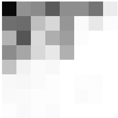
$$\begin{bmatrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 55 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 61 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 76 & 78 & 94 \end{bmatrix}$$

(1)*Original*



$$\begin{bmatrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\ 0 & 0 & -1 & -4 & -1 & 0 & 1 & 2 \end{bmatrix}$$

(2)*DCT*



$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(3)*Quantize*

DCT and quantization

$$
\begin{aligned}
p(u,v) &= \alpha(u)\alpha(v)\sum_{i=0}^{7}\sum_{j=0}^{7} g_{i,j}\cos\left[\frac{\pi}{8}(i+0.5)\,u\right]\cos\left[\frac{\pi}{8}(j+0.5)\,v\right] \\
r(u,v) &= p(u,v)/q(u,v) \\
\alpha(k) &= \begin{cases} \sqrt{1/8} & ,n=0 \\ \sqrt{2/8} & ,n\neq 0 \end{cases}
\end{aligned}
$$

- Use precomputed cosine values (they are expensive and reused).
- Use SIMD instructions that operate on 4 floats.
- Quantization is elementwise division with a quantization matrix.

Main idea

- Each SPE computes one row of 8×8 blocks.
- Input data is `unsigned char` (1byte).
- Output data is `signed short` (2byte).
- Must take care that each SPE gets proper alignment for DMA.
- DCT and quantization is “trivial”, but not when optimizing using SIMD.

Creating SPE contexts

Constructor - excerpt (PPE)

```
spe_argp = (dct_args*) malloc_align(spes*sizeof(dct_args), 16);
dctcontext.resize(spes);

for (int i=0; i<spes; ++i) {
    //Create SPE contexts
    dctcontext[i].speid = spe_context_create(flags, NULL);
    spe_program_load(dctcontext[i].speid, &CellDctQuant_spe);
    dctcontext[i].argp = (void*) &spe_argp[i];
    dctcontext[i].envp = NULL;
    dctcontext[i].entry = SPE_DEFAULT_ENTRY;

    //Set up the arguments for the spe-thread
    spe_argp[i].width = width;
    spe_argp[i].height = height;
    spe_argp[i].padwidth = padwidth;
    spe_argp[i].padheight = padheight;
    spe_argp[i].quanttbl = quantization;
}
```

Running SPE contexts

DctQuantize - excerpt (PPE)

```
while(remainding_block_lines > 0) {
  int spe_threads = min(spes, remainding_block_lines);
  for (int i=0; i<spe_threads; ++i) { //Create threads
    offset_to_block_line = BLOCK_SIZE*(computed_block_lines+i);
    spe_argp[i].input = &(inData[width*offset_to_block_line]);
    spe_argp[i].output = &(outData[padwidth*offset_to_block_line]);

    status = pthread_create(&dctcontext[i].threadid, NULL,
                            &create_spe_thread, &dctcontext[i]);
    remainding_block_lines--;
  }

  for (int i=0; i<spe_threads; ++i) { //Join threads
    pthread_join(dctcontext[i].threadid, NULL);
    computed_block_lines++;
  }
}
```

Arguments to the SPE

CellDctQuant.h - excerpt (SPE/PPE)

```
typedef struct {
  int width __attribute__((aligned(16)));
  int height __attribute__((aligned(16)));
  int padwidth __attribute__((aligned(16)));
  int padheight __attribute__((aligned(16)));

  unsigned char* input __attribute__((aligned(16)));
  signed short* output __attribute__((aligned(16)));
  unsigned char* quanttbl __attribute__((aligned(16)));
} dct_args; //To pass arguments to the SPE
```

Acquiring the argument struct

CellDctQuant_spe.c - excerpt (SPE)

```
int main(unsigned long long speid,
         unsigned long long argp,
         unsigned long long envp) {
  dct_args args __attribute__((aligned(16)));

  ... //misc variable defs.

  lsa = (void*) &args;
  ea = argp;
  size = sizeof(dct_args);
  tagid = 0;
  cmd = MFC_GET_CMD;
  spu_mfcdma32(lsa, ea, size, tagid, cmd);
  spu_mfcstat(MFC_TAG_UPDATE_ALL);

  ...
```

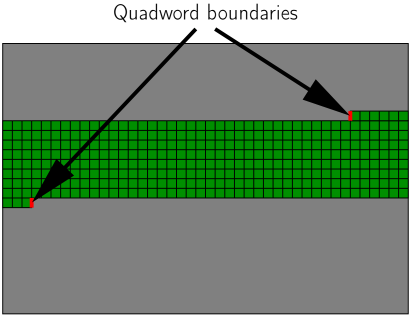
Acquiring the quantization table

CellDctQuant_spe.c - cont excerpt (SPE)

```
...
unsigned char quanttbl[64] __attribute__((aligned(16)));

//The quantization table
size = 64*sizeof(unsigned char);
lsa = (void*) quanttbl;
ea = (unsigned int) args.quanttbl;
tagid = 1;
cmd = MFC_GET_CMD;
spu_mfcdma32(lsa, ea, size, tagid, cmd);
...
```

Acquiring input data

- Width % 16 -> Input alignemnt issues (4 bpp / 1 byte).
- Width % 8 -> Output alignment issues (8 bpp / 2 byte).
  - Outputsize is always % 8, we are lucky:)
- Must ensure start address is aligned, and size is a multiple of quadword.



Acquiring input data (cont.)

CellDctQuant_spe.c - excerpt cont. (SPE)

```
...
unsigned char* input __attribute__((aligned(16)));

offset = ((int) args.input) % 16; //< offset to 16
size = args.width*height  //number of elements
  * sizeof(unsigned char) //bytes per element
  + offset; // our 16-boundary offset
size = (unsigned int) ceil(size/(float) 16)*16; //Pad
void* input_tmp = malloc_align(size, 16);

lsa = input_tmp;
ea = (unsigned int) args.input-offset;
tagid = 2;
cmd = MFC_GET_CMD;
spu_mfcdma32(lsa, ea, size, tagid, cmd);
input = input_tmp + offset;
...
```

Preparing for SIMD

CellDctQuant.h - excerpt (SPE/PPE)

```
typedef union {
  vector float v;
  float s[4];
} v4f; //For SPE intrinsics
```

Precomputing sines using SIMD

CellDctQuant_spe.c - excerpt cont. (SPE)

```
vector float cosuv[8][8][8][2] __attribute__((aligned(16)));

for(v = 0; v < 8; v++) {
  for(u = 0; u < 8; u++) {
    vector float uVec = spu_splats((float) u);
    for(j = 0; j < 8; j++) {
      //v*(2*j+1)*pi/16
      vector float vjVec = spu_splats((float) v*(2*j+1)*pi16);

      //(2*i+1)*pi/16, i=0..3
      vector float iVec1 = {1*pi16, 3*pi16, 5*pi16, 7*pi16};

      //(2*i+1)*pi/16, i=4..7
      vector float iVec2 = {9*pi16, 11*pi16, 13*pi16, 15*pi16};

      //cos(u*(2*i+1)*PI/16)*cos(v*(2*j+1)*PI/16)
      cosuv[v][u][j][0] = spu_mul(cosf4(spu_mul(iVec1, uVec)),
                                  cosf4(vjVec));
      cosuv[v][u][j][1] = spu_mul(cosf4(spu_mul(iVec2, uVec)),
                                  cosf4(vjVec));
    }
  }
}
```

CellDctQuant_spe.c - excerpt cont. (SPE)

```
for (k=0; k<args.width; k+=8) { //For each block
  for (v=0; v<BLOCK_SIZE; ++v) { //For each element in dct block
    for (u=0; u<BLOCK_SIZE; ++u) {
      v4f dctVec;
      dctVec.s[0] = 0; dctVec.s[1] = 0; dctVec.s[2] = 0; dctVec.s[3] = 0;

      for (j=0; j<height; ++j) { //Calculate dct
        v4f inputVec1;
        v4f inputVec2;
        for (i=0; i<4; ++i) { //Shift the values (unsigned->signed)
          inputVec1.s[i] = (float) input[j*args.width+k+i]-128.0f;
          inputVec2.s[i] = (float) input[j*args.width+k+i]-128.0f;
        }
        dctVec.v = spu_madd(inputVec1.v, cosuv[v][u][j][0],
                          spu_madd(inputVec2.v, cosuv[v][u][j][1],
                                   dctVec.v));
      }
      float a1 = !u ? isqrt2 : 1.0f;  float a2 = !v ? isqrt2 : 1.0f;

      output[v*args.padwidth+k+u] = (signed short) ((dctVec.s[1] +
                    dctVec.s[1] +
                    dctVec.s[2] +
                    dctVec.s[3])*a1*a2)/(4.0f*quanttbl[u*BLOCK_SIZE+v]);
    }
  }
}
```

DMAing the result to the PPE

CellDctQuant_spe.c - excerpt cont. (SPE)

```
//DMA the data back to the CPU.
lsa = (void*) output;
ea = (unsigned int) args.output;
size = args.padwidth*8*sizeof(signed short);
tagid = 3;
cmd = MFC_PUT_CMD;
spu_mfcdma32(lsa, ea, size, tagid, cmd);
spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

Summary DCT

- Create SPE contexts in the constructor.
- Use pthreads to start the SPE contexts.
- Issue DMA request for input data.
- Use a C union to couple vector float and float[4].
- Precompute sines using SIMD instructions.
- Use precomputed sines to compute the DCT.
- Quantize.
- DMA result back to the PPE.

Conclusion

- Using DMA can be frustrating (but "easy" after a couple of tries).
- SIMDification of code can be hard, but worth it.
- (Risc of compiler bugs).
- Great speedups (e.g., 25$\times$ faster than CPU for folding@home)
- The Cell processor is versatile (can be used from pipelining to embarassingly parallel problems), and readily available!

Resources

Source code

http://babrodtk.at.ifi.uio.no/Research/Cell

Can be compiled and run on PS3, Cell blade, *or* Cell simulator

Cell BE documentation

http://www.ibm.com/developerworks/power/cell/

Cell BE code samples

/opt/ibm/cell-sdk/prototype/src/samples/