Programming the SPE

Quirin Meyer

March 17, 2006

1 Introduction and Overview

The SPEs, so called Synergestic Processing Units, play an important role in the novel Cell Broadband Engine (CBE) design created by IBM, Sony and Toshiba. The Power Processing Unit, which is a state of the art superscalar RISC processor with a 512K L2 cache, is enhanced by a number of SPEs. In current designs this number is typically eight, but future processors based on the CBE are likely to accommodate more. The basic idea of these SPEs is to offload work from the PPU to them by creating SPE threads that run independently.

In this handout, the overall design of a single SPE is presented. This includes a brief description of the core parts of an SPE, the Synergestic Processing Unit (SPU) and the Memory Flow Controller (MFC). Moreover coding methods are presented and in the last section additional topics are presented. Note that the main reference to this handout is IBM's Cell Broadband Engine Programming Tutorial [1].

2 Hardware Overview

As already mentioned the SPE is subdivided into an MFC and SPU. Architectural, the SPU is a RISC processor with a large register file containing 128 register each 128 bit (i.e. 16 Byte) wide. Operations on those registers are SIMD commands. Beside the RISC core there is a local store (LS) with the capacity of 256KB. More information will be given in the upcoming section. The second part of the SPE is the Memory Flow Controller (MFC). Its basic purpose is to connect the SPE with the rest of the architecture. This is realized through the so called Element Interconnection Bus (EIB). A central role in the overall design plays the DMA controller, which is responsible for moving data from and to its SPE.
Note that SPEs are not meant to run operating systems, which would contradict the design idea of the Cell Processor, which states that the PPE has the main control of the application and subdivides the problem among several SPEs.

3 SPU

The SPU is the number crunching module in the SPE. Its register file with its 128 members is sufficiently large for a load-store architecture. Generally those

registers allow data types sized from 8 Bits up to 128 Bits. Floating point data is support too but in contrast to the PPE, double precission is availalbe on the SPUs, besides single precission. However, the performacne is at least one order of magnitude lower (21.03 GFLOPS against 230.4GFLOPS[5]) when using doubles.
Another difference is that the SPE is not supporting the standardized IEEE 754 format for floating point representation but a modification of it, possibly leading to different numerical results.
However, the underlying instruction set is designed for SIMD usage. SIMD is short for "Single Instruction Multiple Data" and explained as follows. Each of the 128 Bit wide registers can be subdivided into several smaller registers. For example, four single precission floating point variables or two double precission floating point variables can be held in one registers. Operations on these subregisters are performed independently but simultaneously. Therefore the name SIMD becomes quite clear: several variables (the MD in SIMD) get processed by one single instruction (SI). Note that the subdivision of these registers can almost be arbitrary starting from 8 Bits per subregister up to 128 Bits, which means no subdivision at all. Therefore possible data types are: `vector [signed|unsigned] [char|short|int|long|long long]` for integer variables, and `vector float, vector double` for floating point variables. Moreover, the SPE usually performs SIMD operations on all subregister, not only a on subset of them, which makes the exploitation of all subregisters mandatory for achieving maximum performance.
Now that the reservation of a SIMD vector suited for SPE in C/C++ is explained the question of how to access SIMD operations is answered. First of all the compiler can do the job on its own. Moreover, one has the possibility to access the assembly instructions, without writing SPE assembly code, directly by using so called compiler intrinsics. These are architure specific extensions which allow access to a processor instruction set. A nice feature about the usage of intrinsics is that the compiler can still enhance the code by optimization techniques such as loop optimizations, instruction scheduling, data load and store and so forth. Moreover an abstraction to register allocation is provided which is demonstared by the following examples:

```
vector int a, b, c;
/* do something with a,b,c*/
c = si_fa(a,b);
```

This example takes a vector, consisting of four integer variables, adds it to another vector of the same type and stores it into c. Note that this intrinsic maps to one specific assembly instruction. These intrinsics are called **Specific intrinsics**. In practice it is more convenient to use **Generic Intrinsics** which provide the program with data abstraction. For example the intrinsic `spu_add(a,b)` performs different additions depending on the data type of `a` and `b`. This handout is not providing a description of the assembly instructions or the intrinsics. This can be looked up in the documentation[3]. What should however be mentioned is that not all instructions that are availabe on the SPU are implemented on PPU and vice versa. For example the PPE is not supporing double FP and an integer multply and accumlate whereas the SPU has no equivalents for the PPE instructions handling saturating math or logarithms, just to mention a few.

Additionally the SPUs have, in contrast to other processor designs and the PPE, no caches, thus guaranteing constant access times for load and stores.

Before going into to further architectural aspects of the SPU, a little example demonstrating the usage of SIMD commands with intrinsics is given. Consider the squared Euclidean norm of a big vector. Mathematically it is described by

$$\|v\|_2^2 = \sum_{i=0}^{n} v_i^2$$

which can be put into code:

```
float euc(float* v, int n) {
  float acc;
  for(i = 0; i < n; i++)
    acc += v[i]*v[i];
  return acc;
}
```

Note that this piece of code is not exploiting SIMD functionality unless the compiler is recognizing that this code can be vectorized. However, mathematically the sum can be decomposed into four sums, i.e.

$$\|v\|_2^2 = \sum_{i=0}^{n/4} v_{4*i}^2 + \sum_{i=0}^{n/4} v_{4*i+1}^2 + \sum_{i=0}^{n/4} v_{4*i+2}^2 + \sum_{i=0}^{n/4} v_{4*i+3}^2$$

assuming that n is a multiple of 4. In code this looks like:

```
float euc(float* v, int n) {
  float acc0;
  float acc1;
  float acc2;
  float acc3;
  for(i = 0; i < n; i+=4) {
    acc0 += v[i  ]*v[i  ];
    acc1 += v[i+1]*v[i+1];
    acc2 += v[i+2]*v[i+2];
    acc3 += v[i+3]*v[i+3];
  }
  return acc0+acc1+acc2+acc3;
}
```

Instead of using four different float variables one float vector is used and SIMD instructions, i.e.:

```
float euc(vector float* v, int n) {
  vector float acc;
  for(i = 0; i < n/4; i++) {
    acc = spu_fmadd(v[i],v[i], acc0);
  }
  return _sum_across_float4(acc);
}
```

Note that if the vectorsize is not a multiple of four special treatment is necessary for the remaining components.

Notably, the SPU execution units can perform up to two operations simultaneously. Floating point, integer operations and byte operations are executed on the **Even Pipeline** and load and store instructions, branch hints, branch resolutions, channel interface instructions, access to special purpose registers and shuffle instructions are issued on the **Odd Pipeline**. The SPU always executes instructions in program order, in contrast to contempory super scalar architectures no instruction reordering is done ([1], [7]). The disbrution among the two pipelines is implement as follows: After two instructions got fetched from the local store the SPU tries to issue both. If this is not possible, due to pipeline stalls or data dependencies, the first instruction gets issued and the second is issued as soon as possible. Only after both instructions got executed the next instruction pair is fetched. Note that the cycles per instruction (CPI) rate is due to dual issuing normally < 1 ideally 0.5.

When compiling the code of the squared Euclidean norm from above (with `gcc-spu -O3`) using `make filename.s` to retrieve the assembly output, one can see that the inner loop of the routine has been translated to:

```
.L9:
        ai      $5,$5,-1
        lqx     $15,$4,$6
        lqx     $14,$4,$6
        ai      $4,$4,16
        nop     $127
        nop     $127
        fma     $8,$15,$14,$8
.L17:
        brnz    $5,.L9
```

Using the CBE Simulator one can validate that only 25 % of all instructions are performed in parallel and 50 % of the cycles are either spent on stalls or nops. Moreover one fourth of all instructions is issued non parallel. Modifying the code from above by implementing software pipeling, in order to hide load latencies, and loop unrolling [7], 70 % of all instructions are issued in parallel leaving virtually no stalls and nops penalty cycles. Note that still 28 % of all instructions are single issued:

```
vector float temp0 = bigvec[j];
vector float temp1 = bigvec[j+1];
vector float temp2 = bigvec[j+2];
vector float temp3;
for (; j<VECTOR_SIZE-4;j+=4) {
  acc0 = spu_madd(temp0, temp0, acc0);
  temp3 = bigvec[j+3];
  acc1 = spu_madd(temp1, temp1, acc1);
  temp0 = bigvec[j+4];
  acc2 = spu_madd(temp2, temp2, acc2);
  temp1 = bigvec[j+5];
```

```
    acc3 = spu_madd(temp3, temp3, acc3);
    temp2 = bigvec[j+6];
  }
  temp3 = bigvec[j+1];
  acc0 = spu_madd(temp0, temp0, acc0);
  acc1 = spu_madd(temp1, temp1, acc1);
  acc2 = spu_madd(temp2, temp2, acc2);
  acc3 = spu_madd(temp3, temp3, acc3);
```

Alternatively one can use IBM's `spuxlc` which performs way better than `gcc`. Whereas the latter is not performing loop unrolling at all IBM's counterpart unrolls the loops.
Before going into the discussion of MFC, some attributes of the local store should be mentioned. Each load and store operation issued from the exectution unit has a latency of six cycles and 16 bytes per cycle can transfered. In order to guarantee efficient load and store operations, data should always be aligned to a 16 bytes boundary, e.g.

```
vector int eastwood[128] __attribute__ ((aligned(16)));
```

4 MFC

Until now it was assumed that data has already been placed in the local store. The SPU itself does not have a mechanism to access Main Storage (MS) directly. This can only be achieved by using the so called Memory Flow Controller (MFC), which consists of **Memory Mapped IO Registers** (MMIO) and a DMA Controller (DMAC). The primary function of the MFC is to connect the SPU to the EIB in order to issue DMA transfers from or to the LS. The MFC of an SPE is addressed from within the SPE by using channel instructions and from outside the SPE, i.e. other elements such as the PPE or other SPEs, these register are mapped into MS [2].

4.1 DMA

DMA transfers are designed to transport large data portions to and from the SPE. Per cycle up to 128 Bytes can be transported. The maximum number that can be handled by a single DMA transfer is limited to 16KB. In between a multiple of 16 Bytes is mandatory, except byte sizes of 2,4 and 8. Best performance is archived with a 128 Byte alignment for both source and destination addresses and if the transfered size is a multiple of 256 Bytes ([1]). Note that DMA is non blocking, i.e. the execution of the code running on the SPE is not interrupted.
In order to issue more than one DMA command, several of them can be placed within a list. 2048 of these lists are available, each with a maximum size of 16KB. Since DMA commands are non blocking they are stored in two queues, depending from where the command is issued. If the PPE or other elements use the MFC of an SPE, the commands are placed in the MFC Proxy Command Queue, otherwise, the MFC SPU Command Queue is filled by using so called

Channel Instructions.

The following example demonstrates how to initiate a DMA transfer from an SPE. The respective channel registers are written by using the spu intrinsic `spu_writech`. The channels have to be written in a specific order and with the last write instructions the command is issued to the MFC SPU command queue [2],[1].

```
void dma_transfer(volatile void* lsa, unsigned int eah,
                  unsigned int eal, unsigned int size,
                  unsigned int tag_id, unsigned int cmd) {
  spu_writech(MFC_LSA,(unsigned int)lsa);
  spu_writech(MFC_EAH, eah);
  .
  .
  spu_writech(MFC_Cmd, cmd);
}
```

Alternatively `spu_mfcdma64(ls, eah, eal, size, tagid, cmd)` provided by the SDK [1] can be used. `ls` represents the local store address, `eah`, `eal` is the address in main storage, and `size` is the number of bytes the data block possesses. The last parameter `cmd` is the command describing what kind of transfer is to be initiated, which can basically be either `MFC_GET`, which loads data from MS to LS and `MFC_PUT`, which loads data to MS from LS. In order to synchronize commands the `tagid`, ranging from 0 to 31, labels the transfer or transfer groups. This of special interest when using barrier or fences to order commands. For more information refer to [1].

In the following example an array is loaded into LS from MS:

```
/* select all groups to be included in query or wait operations*/
spu_writech(MFC_WrTagMask, 1);

/* start DMA command in order toget data */
spu_mfcdma32((void*)spearray, (unsigned int)ppearray, size, 1,
             MFC_GET_CMD);

/* wait for all dma transfer to be done    */
/* Possible parameters:                    */
/* 0 non-blocking                          */
/* 1 block for any commands to be complete */
/* 2 block for all commands to be complete */
spu_mfcstat(2);
```

Note that `spu_mfcstat(int)` is a mean to wait for DMA transports, whose tagid has been masked by `spu_writech(MFC_WrTagMask, 1)`, to be finished. More information about DMA transfers especially from PPE to SPE can be found at [4], [2] and [6].

4.2 Further communication means

Signal Notification is a mean to sent short 32 Bit message to an SPE. Each MFC has two registers were signals can be written to or read from. Here is a little

example that also introduces the creation of SPU threads on the PPE: First the PPE creates a number of threads and sends signals to all threads.

```
extern spe_program_handle_t signal_spu;
int main() {
  int i, status;
  speid_t spe_ids[SPE_THREADS];
  for (i=0; i<SPE_THREADS; i++)
    spe_ids[i] = spe_create_thread(0, &signal_spu, NULL, NULL, -1, 0);
  for (i=0; i<SPE_THREADS; i++)
    spe_write_signal(spe_ids[i], SPE_SIG_NOTIFY_REG_1, i+42);

  for (i=0; i<SPE_THREADS; i++)
    spe_wait(spe_ids[i], &status, 0);
}
```

The threads created on the SPE do nothing but waiting for the signal to arrive and then terminate.

```
int main(unsigned long long spu_id, unsigned long long parm) {
  int ack;
  unsigned int signal;
  do {
    ack = spu_stat_signal1();
  } while (ack == 0);
  signal = spu_read_signal1();
  fprintf(stderr, "Recieved signal: %d\n", signal);
  return (0);
}
```

Another way to communicate between PPE and SPE are so called mailboxes. Each SPU has an inbound mailbox that, in contrast to signals, can hold more than one entry, namely four. Therefore it is organized as a FIFO queue. Moreover there is an outbound mailbox and interrupt mailbox. Access to mailboxes is once more realized through channel operations from within the SPE and via MMIO Register from the PPE.

5 Coding Methods

5.1 Double Buffering

From the last section it is known that DMA transfers are non-blocking. However to make sure that the data is available the corresponding tagid of the DMA transfer has to be masked by using `spu_writech(MFC_WrTagMask, tagidmask)` and a subsequent `spu_mfcstat(2)` has to be called. This might lead to the following (poor) schematic code when the block being transfered is subdivided into several smaller blocks `B[i]`.

```
Foreach i
  Transfer B[i] from MS to LS
  Wait for B[i]
  Process B[i]
```

Clearly the bottle neck is in line three, because the process waits actively for the data to arrive. A more smarter way is to process a previously transfered buffer while transfering the next buffer, i.e.

```
Transfer B[0]
Foreach i
  Transfer B[i]
  Wait for B[i-1]
  Process B[i-1]
Wait for completion of B[i-1]
Process[n]
```

5.2 Branches

The SPU's execution unit fetches instruction after instruction. In order to be more efficient it even prefetches instruction. Unfortunately when a branch instruction occurs, that jumps to a different location in the code, instructions that have already been prefetched are not usable and the instructions at the new program location have to loaded. This costs about 20 cycles. Compared to the latency of a regular SPE instruction, ranging from 2 - 7 cycles, this is rather large.
Therefore it is wise to avoid unnecessary branching, for example when calling functions. This causes two jumps, one to jump into the routine and one to return from it. Inlining does nothing but placing the code of the subroutine at the location where it is called by the compiler.
Moreover loop unrolling is also a good mean to reduce the number of loops and the number of branches in a `for` statement, for instance. Due to the large number of registers, this is possible. Loop unrolling can be done either manually or by the compiler (either globally by providing a flag, e.g. `spuxlc -qunroll=[yes|no|auto]` or by locally by placing a `#pragma` in front of the according loop).
Jumps due to conditional statements are more harder to treat. On the SPE simple `if-else` statements such as

```
if (a > b)
  d += a;
else
  d += 1;
```

can be replaced by calculating both results and finally select the correct by applying the spu intrinsics to evaluate the condition (here `spu_cmpgt`) and `spu_sel` to select the repsective solution:

```
select = spu_cmpgt(a,b);
d1 = spu_add(d, a);
d2 = spu_add(d, 1);
d = spu_sel(d1, d2, select);
```

This is usually faster than branching.
Additionally to selection, branch hints can placed into the code, indicating the most likely possibility when writing if-statements. The information passed to those hints can either be dynamically, based on a runtime profile, or statically by

a priori knowledge. Branch hits are implemented in C using `__builtin_expect(pred, l))`, where `pred` is the predicate and `l==1` indicates, that the `if` branch is more likely than the `else` branch and `l==0` referees to the inverse situation. Note that branch hints map to assembly instruction and the compiler does its best to place them correctly in the code. Here is an example demonstrating branch hints:

```
if(__builtin_expect(a > b, 0))
  c += a;
else
  d += 1;
```

6 Miscellaneous

6.1 Application Partioning

In this section we want to leave the isolated view on the SPE towards program models that integrate the SPE in the overall architecture. Therefore one distinguish between PPE centric and SPE centric models. In the latter one the PPE plays the role of a centralized resource manager from which the SPEs fetch their jobs one after the other. In the PPE centric model the main application still runs on the PPE while the SPEs complete several task. This can be in a form of a pipeline where data is transfered from one SPE to the next and at each stage a little more of the final solution to the problem is added. An example for this is a computer graphics pipeline.
In parallel stages the problem gets subdivided into several subproblems that can be solved simultaneously, e.g. a matrix multiplication. In a service model functions are distributed among several SPEs and the PPE asks the SPEs to deal with a certain problem, e.g. one SPE does video encoding, the other handles video decoding and the third is good at solving sparse linear systems.

6.2 Porting Code from PPE to SPE

The suggested developing process in [1] is first to write fully vectorized code on the PPE and then port it to the SPEs. This has the advantage that memory is not limited to 256KB, DMA transfers can be neglected and the PPE provides better debugging facilities. After having written an PPE program that has to be ported to the SPE several pitfalls might occur, such as the different SIMD instructions sets, that come along with different performance characteristics. Moreover there are instructions that do not exist either on the SPE but on the PPE or vice versa. Moreover floating point representation is different. One strategy that [1] offers is so called macro translation. This encourages the usage of intrinsics that map 1-to-1 on both instruction sets. Moreover, one should only use data types that are available on the SPU and the PPU and it is wise to construct vectors by applying the same vector literal construtions.

References

[1] *Cell Broadband Engine Programming Tutorial, Version 1.0.* IBM, 2005.

[2] *SPE Runtime managment Library*. IBM, 2005.

[3] *SPU C/C++ Language Extensions*. IBM, 2005.

[4] *Synergistic Processor Unit Instruction Set Architecture*. IBM, 2005.

[5] Chen T. et. al. Cell broadband engine architecture and its first implementation. Technical report, IBM, 2005.

[6] Srinivasan V. et. al. Cell broadband engine processor dma engines. Technical report, IBM, 2005.

[7] Goedecker and Hoisse. *Performance Optimization of Numerically Intensive Codes*. SIAM, 2001.