SPU Programming Tips

Daniel A. Brokenshre

1.0 Background

The SPU (Synergistic Processing Unit) is very powerful computational unit. However, its true power may not be realized unless specific coding practices are applied. This paper attempts to highlight several of the strategies one should employ to achieve optimal performance from your SPU programs.

Please note, some code generators may automatically exploit these techniques without any specific programmer involvement. Understanding the features and limitations of your development tools is important to minimizing your coding effort and maximizing your results.

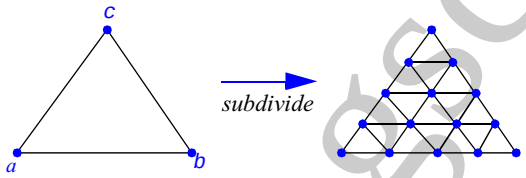
2.0 Programming Tips

2.1 Choose Appropriate SIMD Strategy

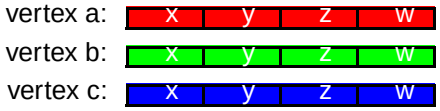
There are multiples ways to SIMD an algorithm. It is important that the chosen method is appropriate for the algorithm, data organization, and local store budget.

Consider, for example, subdivision surfaces in which a single triangle (triangle defined by floating-point vertices a, b, c in Figure 1) is subdivided into multiple triangles. There are at least 3 methods of vectorizing this algorithm.

Figure 1 Triangle Subdivision

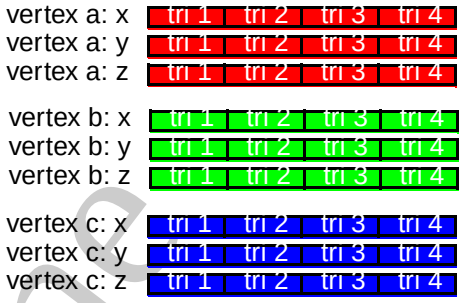


1. *evaluate subdivision vertices one at a time.*
   This is traditional scalar method of performing subdivision. Vertices are typically represented in a "*vec-across*" form (i.e., each 3 or 4 component vertex is maintained in a single SIMD vector).



   This method of representing a 3-D vertex is very natural and often produces small code. However, when applied, it typically produces less efficient code and generally requires significant loop unrolling to improve its efficiency. If the vertices contain less components then the vector can hold (e.g., 3 component vertices), then further efficiencies are compromised.

2. *evaluate subdivision vertices of 4 independent triangles one at a time.*
   An easier method of SIMDing code is to program as if it were scalar and populate the vectors with independent data. Vertex data is represented in "*parallel-array*" format in which each vertex component is maintained in a separate array.



   If the input data is in the *vec-across* format, the spu_shuffle intrinsic can be used to put the data into a *parallel-array* format.

3. *evaluate 4 subdivision vertices for a single triangle at a time.*
   In this case the vertex control data is replicated across the vectors and unique weighting factors are maintain in each element of the vector so that 4 subdivided vertices can be computed in parallel. Inefficiency can result when the number of subdivision vertices is not a multiple of 4.

The SDK contains samples of all three of these SIMD strategies for Point-Normal triangle subdivision. The following metrics (Figure 2 on page 1) were extracted to compare and contrast the performance, code size, and efficiency of each of these techniques. This data shows that the appropriate algorithm depends on the performance objectives and code size constraints.

Figure 2 PN-triangle subdivision metrics

Metric	Strategy				
	1			2	3
Unroll	no	2	4	no	no
Normalized Performance	1.00	1.52	1.91	2.07	1.86
CPI	1.50	1.18	1.02	1.06	0.96
Dual Issue Rate	4.9%	8.7%	12.5%	9.0%	12.3%
Dependency Stalls	34.1%	20.4%	11.2%	10.1%	1.9%
Registers Used	65	101	127	112	106

Figure 2 PN-triangle subdivision metrics

Metric	Strategy				
	1		2	3	
Text Size (bytes)	1152	1920	3968	3588	1856

2.2 Utilize intrinsics

If a programmer writes code in a high-level language (e.g., C or C++), they must rely on compiler technology to auto-vectorize the code in order to exploit SIMD capability of the SPU. However, the flexibility of these languages make it extremely difficult to achieve optimal results. This has resulted in programmers having to resort to using assembly for performance critical sections in order to achieve the results they desire. Writing assembly on the SPU can be a daunting task for large, complicated code because of the large register file and specific issue rules.

A compromise solution is *intrinsics*. Intrinsics are essentially inline assembly formulated in the form of a function call. They provide the programmer explicit control of the instructions used while (unlike assembly) eliminate many of the optimization tasks that compilers are good at. These include:

- register coloring
- instruction scheduling
- data loads and stores
- looping and branching
- literal vector construction

For a complete description of the SPU intrinsics, consult the SPU C/C++ Language Extensions specification.

2.3 Understand the SPU's instructions set

When programming the SPU, whether using vector intrinsics or simple scalar code, it is essential that one understand the SPU instruction set. This includes:

- Understanding the mapping of intrinsics to specific instructions. For example: when promoting a scalar to vector, promote the scalar to the "preferred" slot to avoid the introduction of instructions.
- Understanding the dynamic range and sign of immediate values. This can eliminate possible extraneous vector literal constructions.
- Understanding the branch instructions to formulate conditionals that can be efficiently generated.
- Understanding the instructions used to construct (vector) literals so as to reduce the code size associated with their construction.

2.4 Understand issue rules and latencies

Fine tuning an implementation requires and understanding of the issues and latencies in order to maximize the dual issue rates (minimize the Cycles Per Instruction) and reduce dependency stalls.

The SPU has two instruction pipelines - pipe 0 and pipe 1. Instructions are executed on their predefined pipeline (see Figure 3 and Figure 4 instruction pipeline overviews). Each clock, up to 2 instructions can be issued as long as the following rules are observed:

- Instructions must be ordered such that a pipe 0 instruction is followed by a pipe 1 instruction.
- The pipe 0 instruction must reside on an even address (an address whose least significant 3 bits is 000).
- The pipe 1 instruction must reside on an odd address (an address whose least significant 3 bits is 100).
- All operands to the 2 instructions must be available - no dependency stalls.

The following tables provide an overview of:

- which pipeline the instructions are issued on
- the number of clock stalls - the number of additional cycles before another instruction of the same type can be issued. For example, double precision floating operations have a 6 cycle stall. Therefore, for back to back double precision floating point operations, the second operation will be issued at least 7 cycles after the first operation.
- the latency of the instruction - the number of instructions before the result is available.

Figure 3 Pipeline 0 Instructions and Latencies

Pipe 0 Instructions	Stall (clocks)	Latency (clocks)
Single precision floating-point operations	0	6
Integer multiplies, convert between float/integer, interpolate	0	7
Immediate loads, logical ops, integer add/subtract, signed extend, count leading zeros, select bits, carry/borrow generate	0	2
Double precision floating-point operations	6	7
Element rotates and shifts	0	4
Byte operations - count ones, abs difference, average, sum	0	4

Figure 4 Pipeline 1 Instructions and Latencies

Pipe 1Instructions	Stall (clocks)	Latency (clocks)
Shuffle bytes, quadword rotates and shifts	0	4
Loads/stores, branch hints	0	6
Branches	0	4
Channel operations, move to/from SPRs	0	6

2.5 Avoid using external scalars

The SPU only loads and stores a quadword at a time. As such, scalar (sub-quadword) loads and stores require numerous instructions and have long latencies. This is due to the fact that scalar loads most be rotated into the preferred vector slot and stores require a read, scalar insert, write operation. This overhead is demonstrated by the following code sample.

Figure 5 Scalar load/store example

```
SOURCE
void add1(int *p) { *p += 1; }
ASSEMBLY
add1:
    lqd     $4, 0(p)        # load qword @ p addr
    rotqby  $5, $4, p       # move *p to element 0
    ai      $5, $5, 1       # add 1
    cwd     $6, 0(p)        # gen insert shuffle
    shufb   $4, $5, $3      # insert scalar in qword
    stqd    $4, 0(p)        # store updated qword
```

There are several strategies for making scalar code (code that is not appropriate for vectorization) more efficient. These include:

- Change the scalars to quadword vectors. This may seem wasteful, however, if you consider the 3 extra instructions associated with loading and storing scalars, this trade-off can actually have a positive impact on code size.
- Cluster scalars into groups and load multiple scalars at a time using a quadword memory access. Manual extract or insert the scalars on an as needed basis. This will eliminate redundant loads and stores.

The SDK contains an implementation of RC4 - a character base encryption algorithm that utilizes a 256-byte dynamic state table. Because each iteration of the algorithm is dependent upon previous iteration, it can not be parallelized. However, by exploiting the strategies outlined above, significant performance enhancements can still be achieved. The 256 byte state table is expanded into a 256 quadword state table in which all 16 element of the unsigned character vector contain the same byte value. The input and output messages are fetched and stored a quadword at a time to eliminate the extraneous loads and stores and their respective latencies. These changes resulted in a 86% improvement in performance.

Figure 6 RC4 Scalar Improvements

Implementation	Instructions	Cycles	CPI	speedup
original	540120	723245	1.34	-
optimized	265794	388457	1.46	1.86

2.6 Remove, Unroll, and Pipeline Loops

Loops are the foundation in nearly all programs (especially steaming applications). If the number of loop iterations is constant, then consider removing the loop altogether. Otherwise (the number of loop iterations is variable), consider unrolling the loop if the loop is relatively independent (i.e., an iteration of the loop is not dependent upon the previous iteration). The SPU has a large register file and significant loop unrolling can be accomplished before register spill occurs. Register spilling occurs when the instantaneous number of active variables exceeds the size of the register file. Unrolled loops provide additional computation for the optimizer to improve issue rates and reduce dependency stalls.

When unrolling loops, additional local variables may be required to eliminate "false dependencies" amongst the loop variables. Failure to eliminate these false dependencies can cause unrolled loops not to be interleaved by the compiler.

The SDK contains a sample workload, called xformlight, that performs basic graphics vertex processing, 4 vertices at a time. The vertex processing includes 4x4 vertex transformation, perspective division, 1 local light computation with OpenGL style specularity, color conversion, clamping and RGBA color packing. Figure 7 shows how loop unrolling affects the performance of this workload. The loops are unrolled by 2, 4, and 8. In addition, the compiler's automatic loop unrolling was applied. As can be seen, manual loop unrolling is the most affective.

Figure 7 Loop unrolling the xformlight workload

Metric	Unroll Factor				
	1	2	4	8	auto[a]
Normalized Perf.	1.00	1.57	1.69	1.73	1.57
CPI	1.35	0.88	0.78	0.64	0.87
Dual Issue Rate	3.3%	20.5%	33.1%	59.4%	21.6%
Dependency Stalls	27.2%	3.5%	0.7%	1.2%	1.9%
Resource Conflicts	1.1%	1.7%	0.5%	0.0%	1.7%
Registers Used	79	105	128	128	96
Text Size (bytes)	1988	2372	3204	5380	2564

a. automatic loop unrolling results were measured using spuxlc by including the compilation flag "*-qxflag=unroll_large*".

Most loops generally have the same basic structure. Per iteration they load input data, perform computation, and finally store the results. Since loads, stores, quadword rotates, and shuffles execute on pipeline 1, and most computation instructions execute on pipeline 0, loops can be software pipelined to improve dual issue rates by computing at the same time as loading and storing data. Figure 10 shows the results of software pipelining the xformligght workloads.

Figure 8 Basic Loop

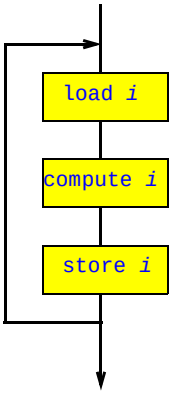


Figure 9 Software Pipelined Loop

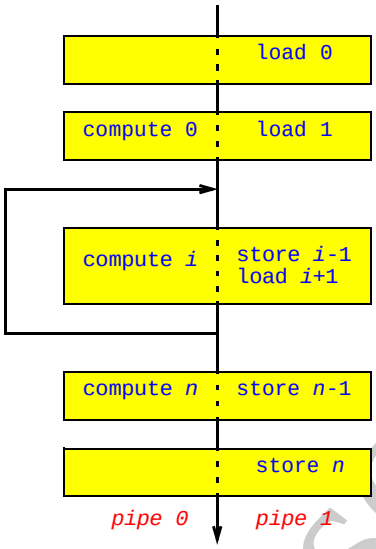


Figure 10 Software pipelined xformlight

Metric	Unroll Factor	
	2	4
Normalized Perf.	1.82	1.83
CPI	0.75	0.69
Dual Issue Rate	36.7%	47.8%
Dependency Stalls	1.3%	0.5%
Resource Conflicts	1.0%	0.5%
Registers Used	114	128
Text Size (bytes)	2436	3468
Speedup vs. non-pipelined	1.16	1.08

Note: Applications using auto-vectorization technology should not explicitly remove, unroll, or pipeline loops. This only adds complexity to the auto-vectorization process.

2.7 Use offset pointers

The PowerPC processor supports *load/store with update* instructions. These instructions allow one to sequentially index through an array without the need of additional instructions to increment the array pointer.

The SPU does not support this instruction form. Instead, one should exploit the d-form by specifying small literal array offsets from the base array pointer. For example, consider the following code that has been written to exploit the *store with update* instruction.

```
#define FILL_VEC_FLOAT(_q, _data)                    \
    *((vector float)(_q++) = _data;

FILL_VEC_FLOAT(q, x);
FILL_VEC_FLOAT(q, y);
FILL_VEC_FLOAT(q, z);
FILL_VEC_FLOAT(q, w);
```

The same code snippet can be improved for SPU execution as follows.

```
#define FILL_VEC_FLOAT(_q, _offset, _data)           \
    *((vector float)(_q+(_offset)) = _data;

FILL_VEC_FLOAT(q, 0, x);
FILL_VEC_FLOAT(q, 1, y);
FILL_VEC_FLOAT(q, 2, z);
FILL_VEC_FLOAT(q, 3, w);
```

2.8 Interleave independent code

Some compilers exploit a very small optimization window. As such, unrolling large loops may not achieve optimal performance because the compiler fails to effectively interleave the unrolled loops. In this case, explicitly interleaving the unrolled loops is in order.

2.9 Reduce branching

Branches are expensive. Not only are they expensive from their issuance and stalls due to mis-predicts, they create a boundary for optimization. Therefore, one should avoid their use, if at all possible.

The secret to eliminating branches is exploiting the *select bits* instruction. For example, an if-the-else statement can be made branchless by computing the results of both the *then* and *else* clauses and using *select bits* to choose the result as a function of the conditional. For example:.

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

can be made branchless as follows:

```
select = spu_cmpgt(a, b);
c_plus_1 = spu_add(c, 1);
a_plus_b = spu_add(a, b);
c = spu_sel(c, c_plus_1, select);
d = spu_sel(a_plus_b, d, select);
```

2.10 Reduce branch mis-predicts.

When branches can not be eliminated, then reduce the number of branch mis-predicts. This can be accomplished by either utilizing feedback directed optimization technologies or programmer directed branch prediction. The programmer can explicitly direct branch prediction using the *__builtin_expect* language extension. Considering the previous example, one can direct the compiler that *a* is more likely not larger than *b* thereby preferencing the else condition by adding a *__builtin_expect* as follows.

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

2.11 Avoid full integer multiplies.

The SPU contains only a 16x16 bit multiplier. Therefore, to perform a 32-bit integer multiply, it takes five (5) instructions - 3 16-bit multiplies and 2 adds to accumulate the partial products.

To avoid extraneous multiply cycles, the following rules should be observed.

- If the operands are less than 16 bits in size, cast them to unsigned shorts prior to multiplication to take advantage of the native multiplier.
- Make sure to also cast constants since they have an implicit type of int.
- Keep array elements a power-of-2 sized to avoid multiplication when indexing.
- To avoid inadvertent introduction of signed extends and masks due to casting, consider introducing a macro to explicitly perform an integer multiply whose operands are 16-bits or less.

```
#define MULTIPLY(a, b)   \
    (spu_extract(spu_mulo((vec_ushort8)spu_promote(a,0),
                          (vec_ushort8)spu_promote(b, 0)),0))
```

2.12 Inline functions

Because the SPU has a large register file, many variables can be “alive” at once. This causes function calls to be expensive since volatile registers must be saved and restored. Therefore programmers are encouraged to inline external functions, especially small functions.

Note: For some conditions, inline functions may not prove to have acceptable result and macros may have to be used.

2.13 Replace structures with local variables

Some compilers have problems keeping structure elements in local registers. When this happens, programmers must resort to explicit local variables for each structure element.

2.14 Initiate DMA transfers from the SPU

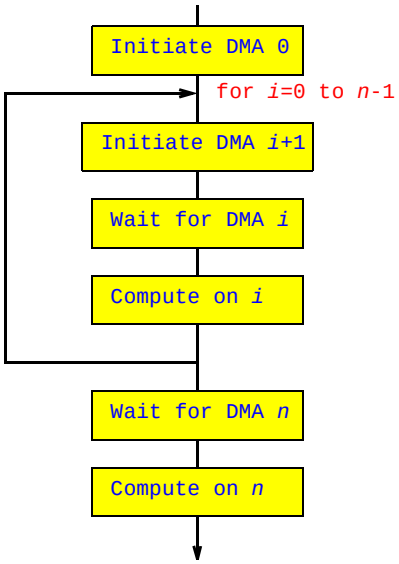
To keep the SPU fed, avoid utilizing the programming model in which the PU pushes data to the SPUs. Instead, have the SPU feed themselves by initiating DMAs from the SPU. This is done because:

- there are more SPUs than PU in a system.
- the PU can only have 4 (plus 2 for L2 loads and stores) in flight data references. Whereas the SPU can have up to 16 DMAs in flight.

2.15 Utilize double buffering

In order to hide data access latencies, utilize double (or multi) buffering techniques. One can either double buffer the data (typical) or double buffer the code, depending upon the code and data sizes and data access patterns.

Figure 11 Double Buffer Loop



When double buffer, the following general rules should be applied.

- Use multiple local store buffers.
- Use unique DMA tag IDs. One for each multi-buffer.
- Use *fence*d commands to order the DMAs within a tag group.
- Use *barrier* commands to order DMAs within the queue.
- Keep array elements a power-of-2 sized to avoid multiplication when indexing.

2.16 Design for Limited Local Store

The SPU local store is a limited resource. Only 256 Kbytes is available for program, stack, local data structures and DMA buffers. Many of the optimization techniques presented in this paper adds additional pressure on this limited resource. As such, all optimization may not be possible for a given application. Therefore, programmers may be forced to utilize only a few optimizations due to the limited local store constraint.

Often it is possible to reduce local store pressure by dynamically managing the program store using code overlays - also known as plug-ins.

3.0 Summary

The SPU is a very powerful processing complex. Much of the power can not be realized unless specialize programming techniques are employed. Utilizing the techniques outlined in this paper will go along way in achieving the full potential of the SPU.

4.0 References

1. “SPU C/C++ Language Extensions”, SCEI/Toshiba,IBM.
2. “Curved PN Triangles”, Alex Vlachos, Jorg Peters, Chas Boyd, Jason Mitchell.
3. “Synergetic Processing Unit (SPU) Architecture Specification”, SCEI/Toshiba, IBM.