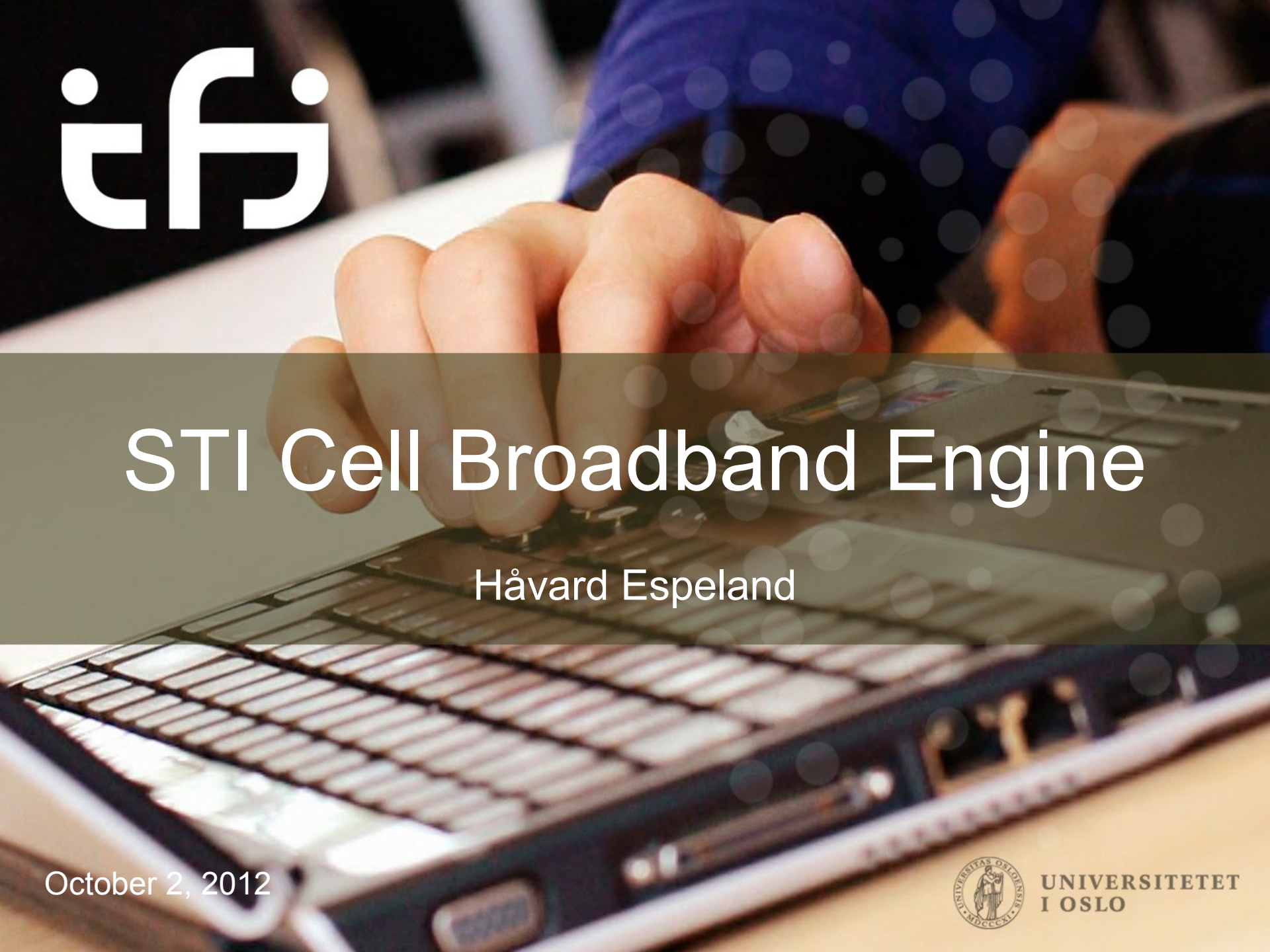


STI Cell Broadband Engine

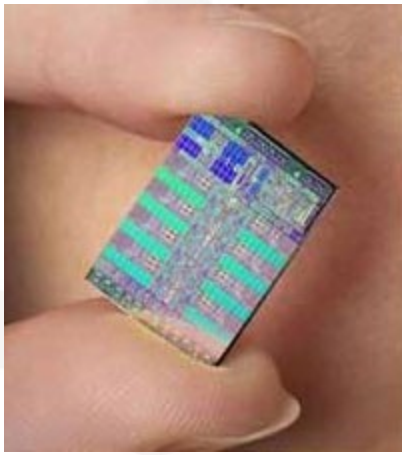
Håvard Espeland

October 2, 2012

UNIVERSITETET I OSLO


RED DEAD
REDEMPTION

Cell Broadband Engine



- The world's 19th fastest supercomputer “Roadrunner” uses 12 960 Cell processors and 6 480 Opteron processors
- Heterogeneous architecture
  - Hard to utilize / Impressive performance
- Processor in PS3 with more than 63 million units sold worldwide.

Performance

- Theoretical single-precision performance of 25.6 GFLOPS on each SPE single precision.
  - 98% of this peak performance achieved for matrix multiplication benchmark (IBM)
- PowerXCell (2008) variant with 12.8 GFLOPS per SPE double precision with a 102.4 GFLOPS total.
- Intel i7 – 980X, 6 cores (2010) – 107.55 GFLOPS

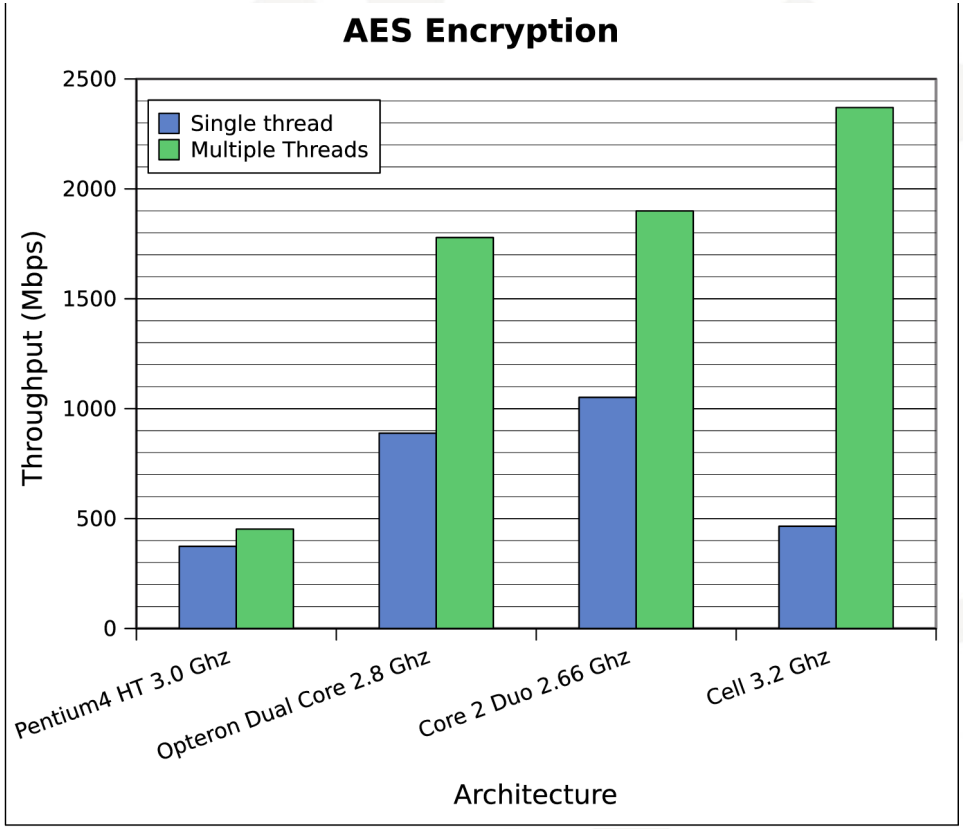
Memory Bandwidth

- Cell XDR bandwidth: 25.6 GB/s
- Intel i7 980X memory bandwidth: 25.6 GB/s
- Theoretical 25.6 GB/s per channel on EIB. Total bandwidth is 204.8 GB/s
  - 197 GB/s total throughput achieved between cores on EIB (IBM)

Power Consumption

- The simple cores use very little power
- 4th generation PS3 system uses up to 60W including disk, GPU, RAM, etc. PowerXCell uses 90W.
- i7-980X CPU alone uses up to 130W (TDP)
  - A complete system draws 200-300W under load (150W idle)

AES Performance



Folding at Home Statistics (Nov 2010)

OS Type	x86 TFLOPS	CPUs	GFLOPS / CPU
Windows	209	220091	0.95
OSX / PPC	4	4477	0.89
OSX / Intel	22	7176	3.07
Linux	63	36784	1.71
ATI GPU	686	6372	107.66
NVIDIA GPU	2112	8415	250.98
PLAYSTATION	1692	28457	59.46
Total	4788	311772	15.36

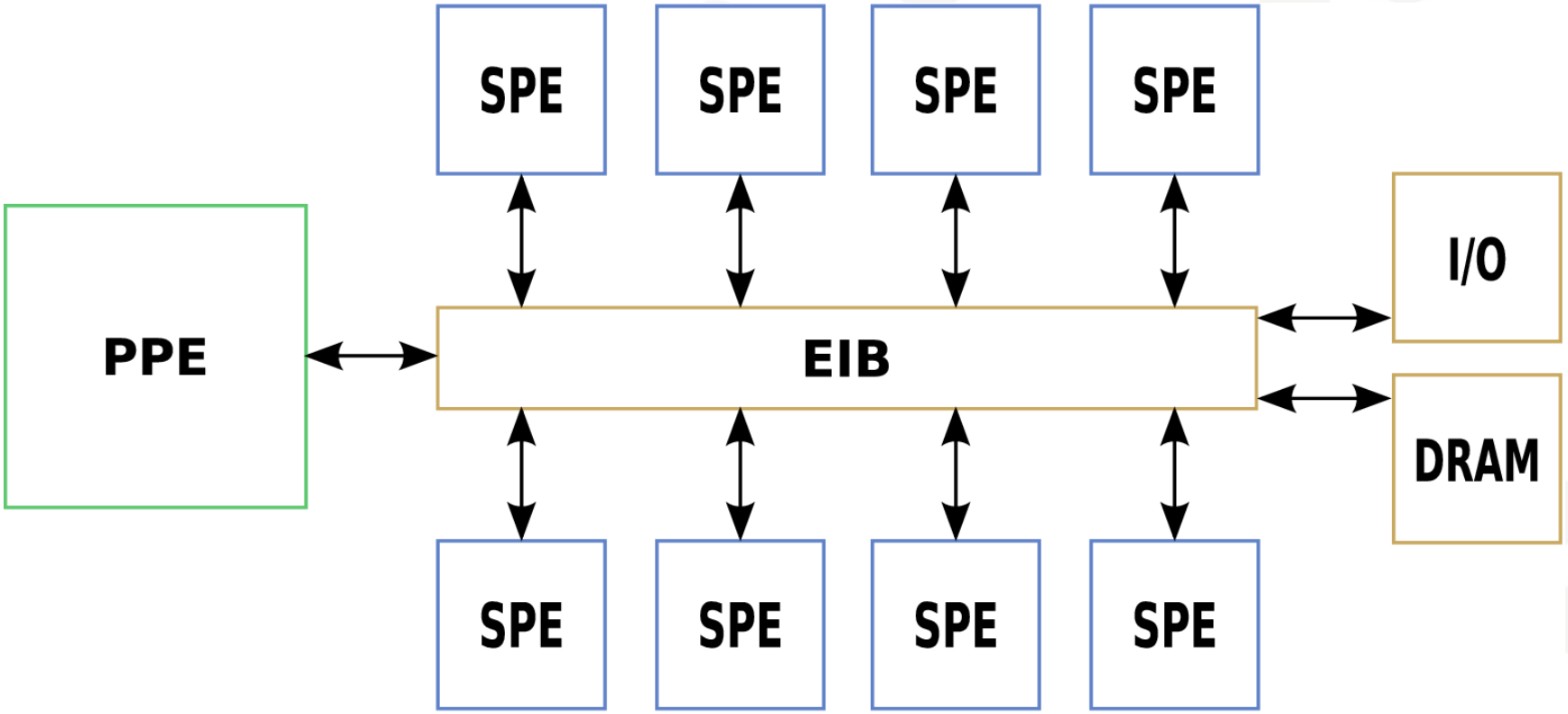
PlayStation 3 used to hack SSL, Xbox used to play Boogie Bunnies

by Joseph L. Flatley posted Dec 30th 2008 at 5:41PM

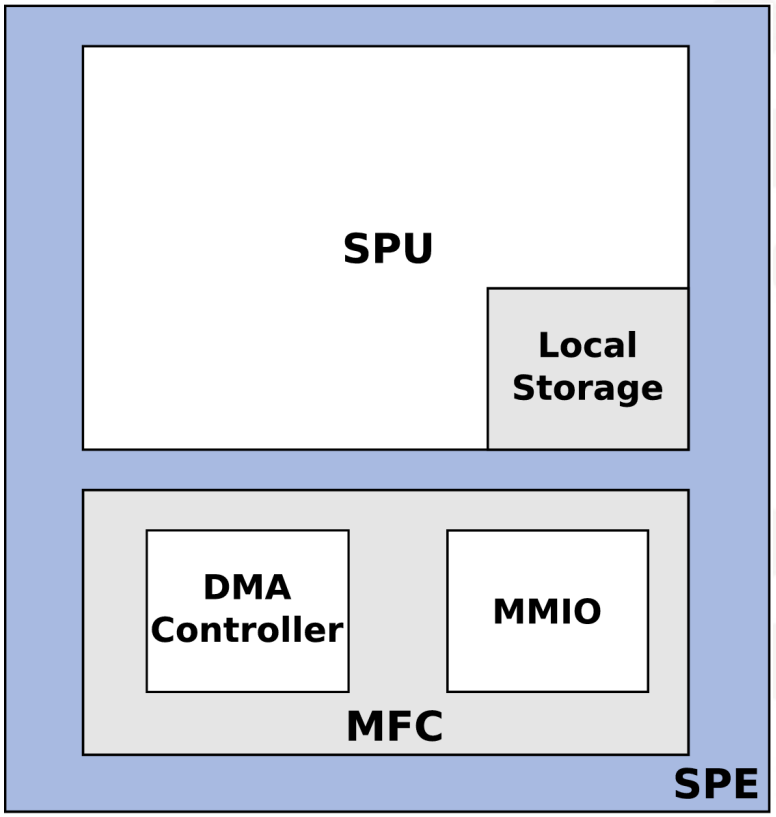
Between the juvenile delinquent hordes of PlayStation Home and some lackluster holiday figures, the PlayStation has been sort of a bummer lately, for reasons that have nothing to do with its *raison d'etre* -- gaming. That doesn't mean that the machine is anything less than a powerhouse -- as was made clear today when a group of hackers announced that they'd beaten SSL, using a cluster of 200 PS3s. By exploiting a flaw in the MD5 cryptographic algorithm (used in certain digital signatures and certificates), the group managed to create a rogue Certification Authority (CA) which allows them to create their own SSL certificates -- meaning those authenticated web sites you're visiting could be counterfeit, and you'd have no way of knowing. Sure, this is all pretty obscure stuff, and the kids who managed the hack said it would take others at least six months to replicate the procedure, but eventually vendors are going to have to upgrade all their CAs to use a more robust algorithm. It is assumed that the Wii could perform the operation just as well, if the hackers had enough room to spread out all their Balance Boards.

[Via ZD Net]

Cell Broadband Engine



Synergistic Processing Element



- 8 SPEs available
- PS3 has 6 (7) SPEs and runs at 3.2 GHz
- 256 kB Local Storage
- Vector processor with 128 128-bit registers
- Hardware channels for communication and synchronization

Synergistic Processing Element

- The SPU can only access local storage, and must use DMA operations to load data from main memory
- DMA operations must follow strict alignment rules
- Cache lines can be locked to support atomic operations used in synchronization primitives
- In-order execution model, and no hardware support for dynamic branch prediction

SPE Limitations

- Branch misses are very expensive (18 or 19 cycles)
- Most instructions process SIMD data and alignment
- Requires explicit DMA operations
- DMA operations should be cache-line aligned (128 B), and not in the PPE's cache
- Very small local storage
- No hardware division

Floating point Limitations

- Slow double precision floating point performance, although this was fixed in the PowerXCell 8i version used in Roadrunner.
- Single precision calculations are optimized for games and not accuracy.
  - Extended range by removing NaN and Infinity
  - Rounded numbers always truncated down
- Not fully IEEE 754 compatible!

SPE Pipeline Considerations

- Each SPE has two instruction pipelines
  - They can execute in parallel if you schedule instructions correctly
- The pipelines have different capabilities, and instructions of different types are dispatched to a specific pipeline

Pipeline 0

Instruction Type	Latency	Stall
Single-precision floating-point operations	6	0
Integer multiplies, convert between floating-point and integer, interpolate	7	0
Immediate loads, logical operations, integer add and subtract, signed extend	2	0
Element rotates and shifts	4	0
Byte operations (count ones, absolute difference, average, sum)	4	0
Double-precision floating-point operations		
Cell/B.E. processor	7	6
PowerXCell 8i processor	9	0

Pipeline 1

Instruction Type	Latency	Stall
Shuffle bytes, quadword rotates, and shifts	4	0
Gather, mask, generate insertion control	4	0
Estimate	4	0
Loads	6	0
Branches	4	0
Channel operations, move to/from special purpose registers (SPRs)	6	0

In-order execution

- Instructions can be executed every cycle if dependencies are satisfied
- Typical x86 CPUs re-order instructions to avoid pipeline stalls. SPEs don't!
- Considers re-ordering your instructions to avoid pipeline stalls.

PPE Multiplication and division

- **mulli** – inserts a 6-cycle stall.
- **mullw, mullwo, mulhw, mulhwu** – inserts a 9-cycle stall.
- **mulld, mulldo, mulhd, mulhdu** – inserts a 15-cycle stall.
- **divd, divdu, divdo, divduo** – inserts a 10 to 70 cycle stall.
- **divw, divwu, divwo, divwuo** – inserts a 10 to 70 cycle stall.
- **fdiv, fdivs, fsqrt, fsqrts, mtvscr** – inserts a minimum 13 cycle stall.

Only 16 x 16 bits multiplier. For larger operations, several subsequent operations are performed.

Communication

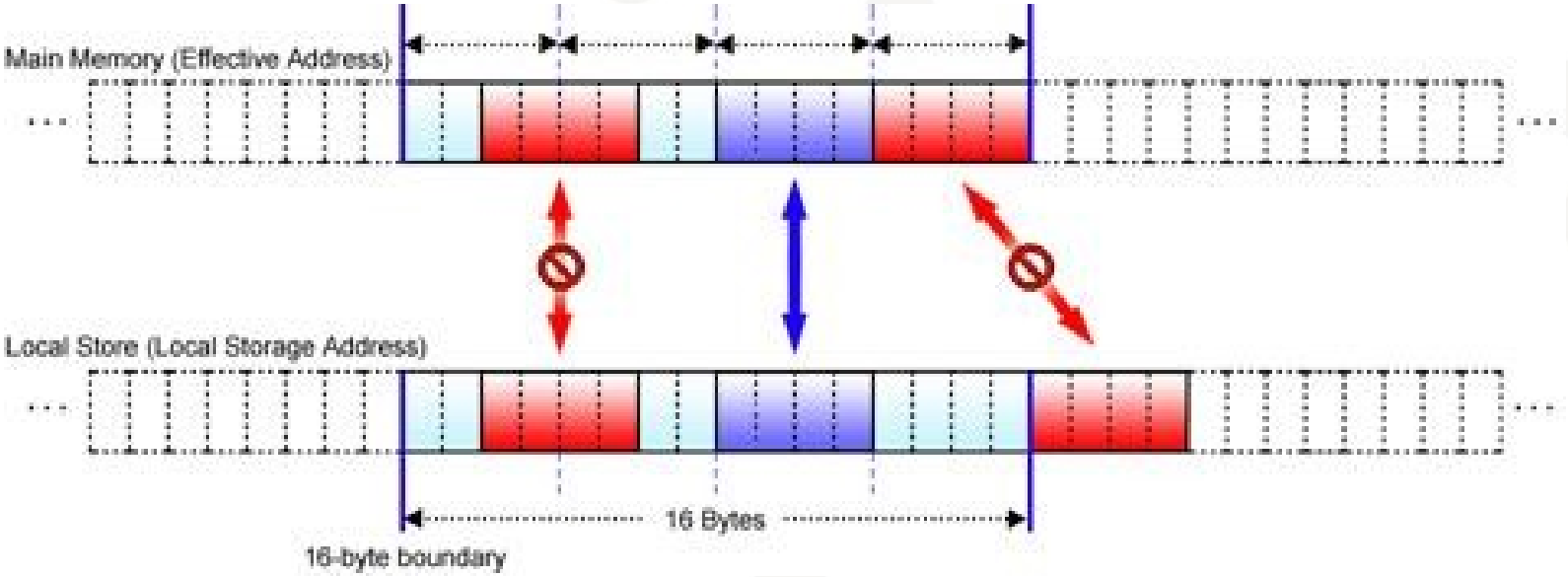
- Dedicated hardware channels available
- Signals
- Mailboxes with (short) queues. Can both interrupt receiver and work in poll-mode.
- Use the primitives to design your own scheme together with DMA operations.

DMA operations

- Explicit commands to transfer from main memory to local storage or between SPEs local storage. DMA operations must follow strict alignment rules
- Transfer size must be a multiple of 16 bytes, up to a maximum of 16 kB. Transfers of 1, 2, 4 or 8 bytes are allowed.
- Source and destination address must be aligned on a 16 byte boundary when the transfer size is >= 16 B

DMA operations

- For transfers < 16 bytes, addresses are aligned on transfer size byte boundary
- Lower 4 bits of src and dst must be equal



DMA operations

- Failure to adhere alignment rules result in

  Bus error!

- Hard to debug since your alignment may only be off occasionally.
- A useful tool for debugging is to wrap DMA functions in macros that prints addresses

Mitigate DMA operation complexity

- Only use multiples of 16 bytes transfers
- Avoid using dynamic memory on SPEs (malloc/memalign). Amount of LS used can then be inspected using spu-objdump.
- On SPEs, use 16 or 128 byte aligned structures and buffers, and put them in .bss
- On PPE, using memalign() is fine.

Programming the Cell

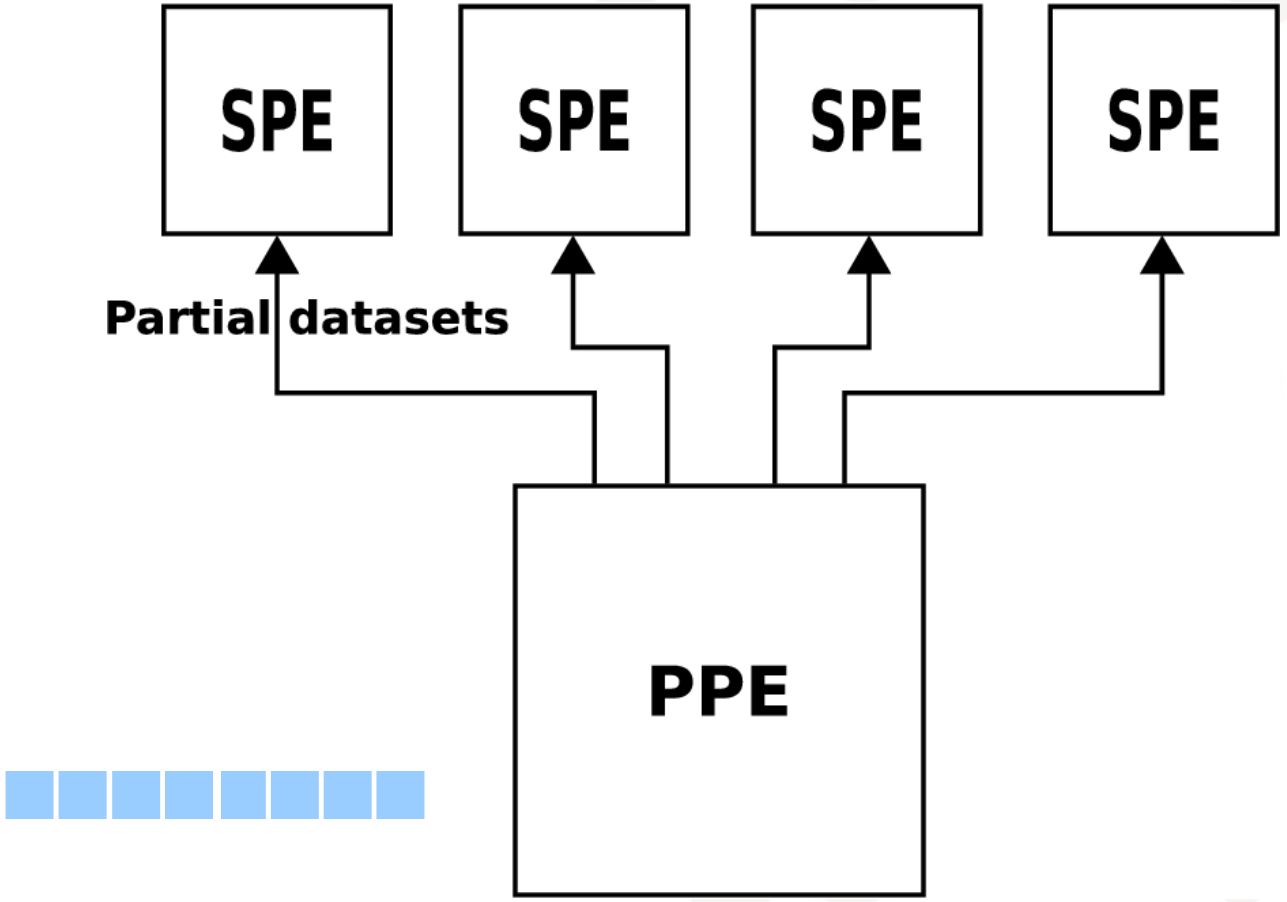
Håvard Espeland

UNIVERSITETET
I OSLO

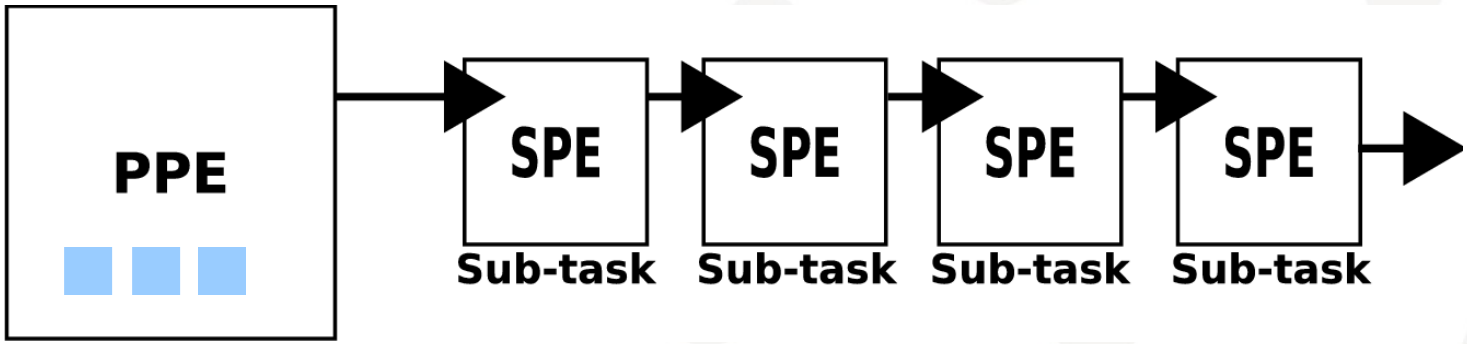
Parallelization Schemes

- Strategies for utilizing the cores available
- Different schemes for different problems
- Two key points to consider:
  - Memory flow of your application
  - Avoid idle cores!

Data Decomposition Strategy



Pipelining Strategy



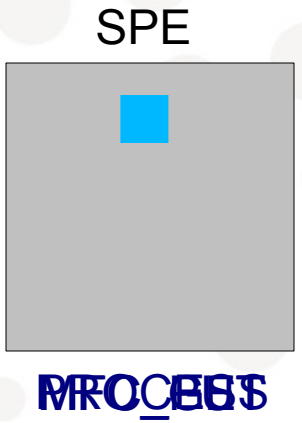
Many levels of parallelism on Cell

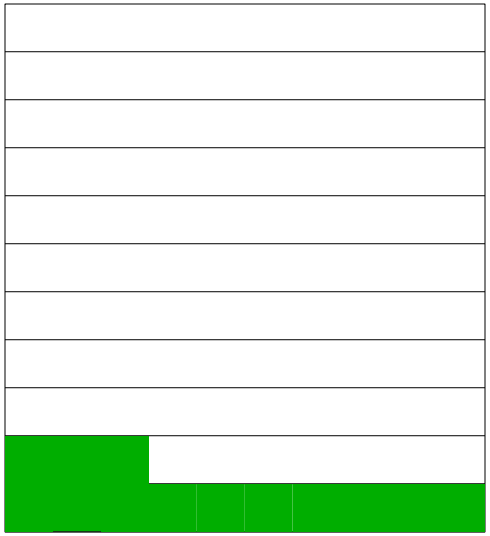
- Instruction level: Two pipelines on each SPE
- Data level: SIMD instructions
- Task level: Many SPEs available

Cell Performance Considerations

- Make sure the cores never wait for work
  - Consider your parallelization strategy
  - Double buffering
- Avoid branches on SPE
  - Many branches can be avoided by computing both solutions and selecting the correct result.
  - Unroll loops
- The SPEs are vector processors; *Use SIMD!*

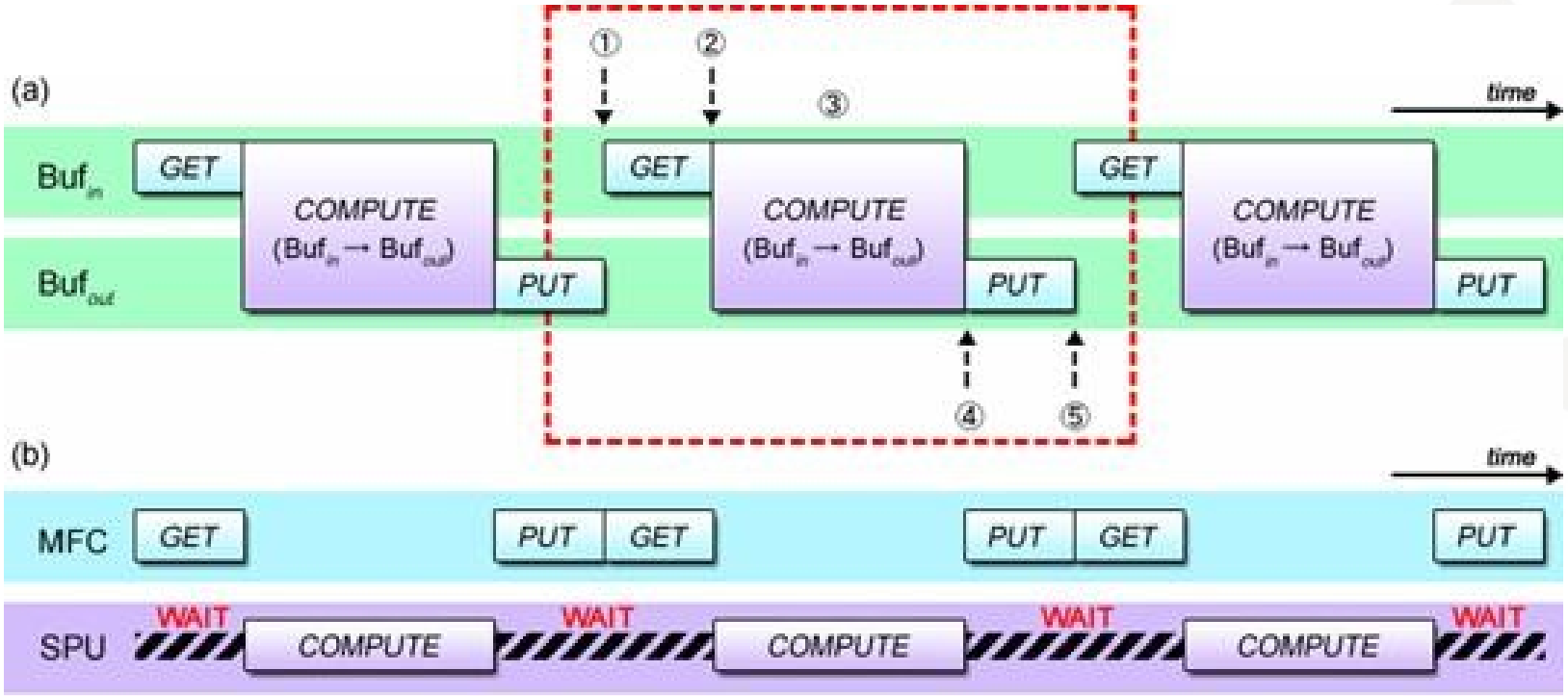
Single Buffering



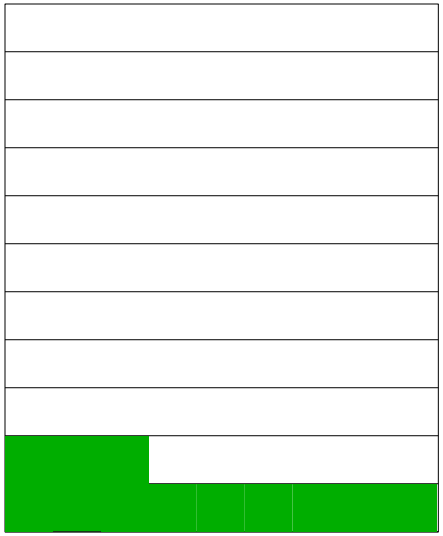


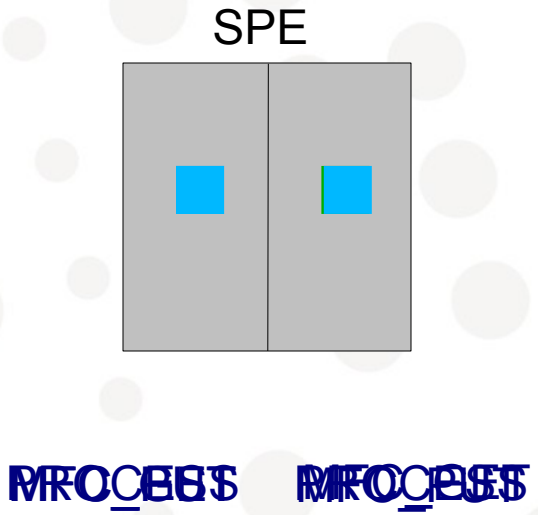
Main Memory

Single Buffer

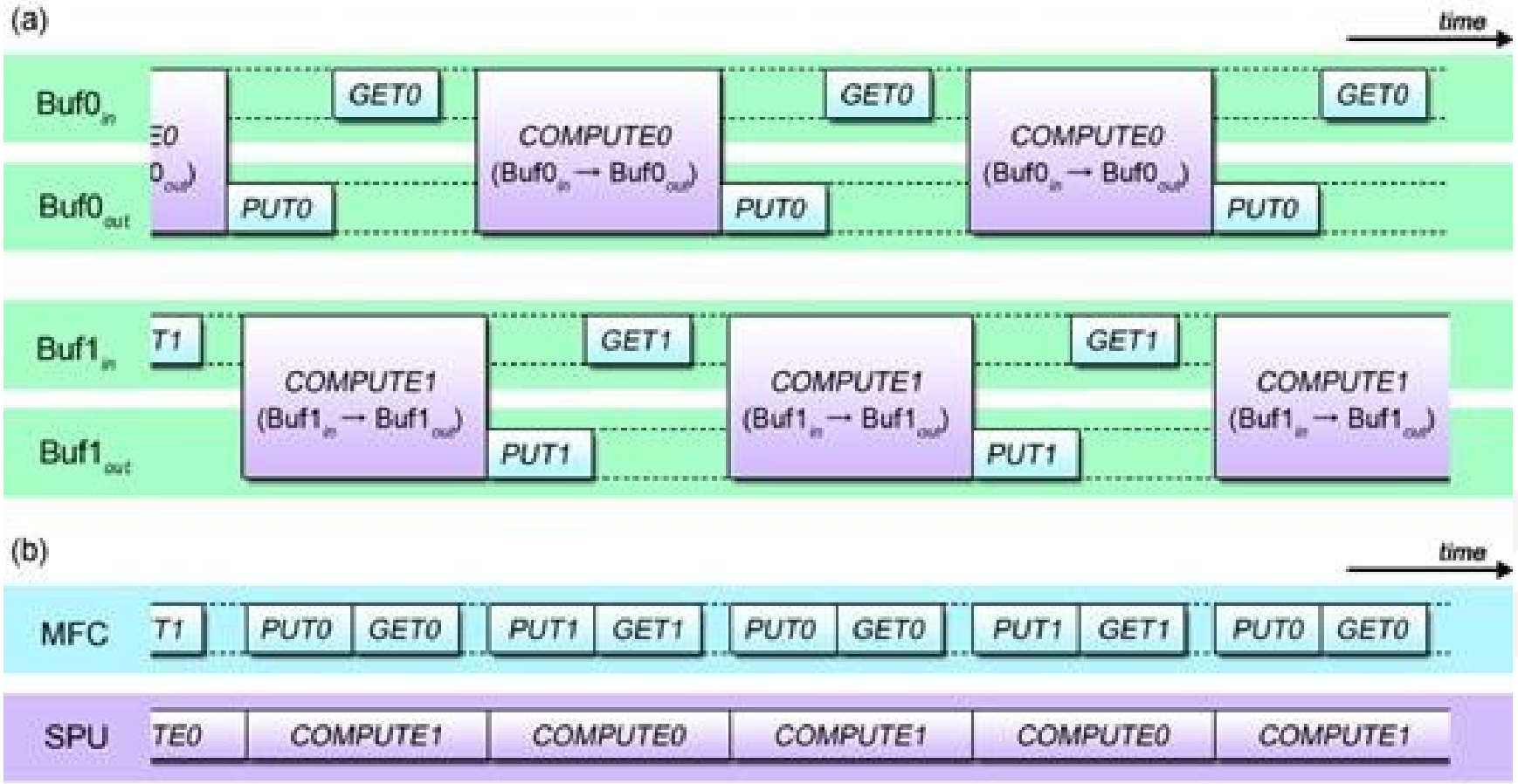


Double Buffering





Double buffering



Development tools

- Gcc has full Cell target support (spu-gcc, ppu-gcc)
- SPEs can be debugged (spu-gdb), and to determine amount of LS used – use spu-objdump
- SPE images can be linked to main executable using ppu-embedspu

Trivial SPE example

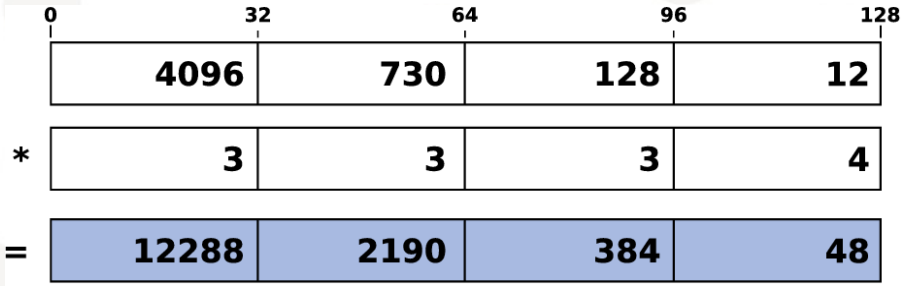
```
#include <spu_intrinsics.h>
#include <stdio.h>

int main(unsigned long long spe, unsigned long long argp,
unsigned long long envp)
{
        __vector float a = {4096, 730, 128, 12};
        __vector float b = {3, 3, 3, 4};

        __vector float result = spu_mul(a, b);

        float first_value = ((float*)&result)[0];

        printf("First_value: %f\n", first_value);
}
```



Spawning an SPE context

```
#include <libspe2.h>
int main(int argc, char **argv)
{
    spe_context_ptr_t spe;
    spe_program_handle_t *prog;
    unsigned int entry;
    spe_stop_info_t stop_info;

    /* Load SPE image */
    prog = spe_image_open("mul_test.elf");

    /* Load and run SPE context */
    spe = spe_context_create(0, NULL);
    int ret = spe_program_load(spe, prog);
    entry = SPE_DEFAULT_ENTRY;
    spe_context_run(spe, &entry, 0, NULL, NULL, &stop_info);

    /* Clean up */
    spe_context_destroy(spe);
    spe_image_close(prog);

    return 0;
}
```

Issuing DMA commands

- Should be issued on SPE, not PPE
- Runs asynchronously and is assigned to a tag group.

```
spu_mfcdma32(ls, ea, size, tagid, cmd)
spu_mfcdma64(ls, eahi, ealow, size, tagid, cmd)

/* cmd is typically MFC_GET_CMD or MFC_PUT_CMD */
```

- Wait until tag group finishes

```
spu_writech(MFC_WrTagMask, 1 << tagid);
spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

Dealing with 32/64 bit

- SPEs use 32 bit addressing, but the PPE supports both 32 and 64 bits.
- May cause confusion when dealing with shared pointers between PPE and SPEs.
- **Solution 1**: Use a Makefile and specify whether you want 32 or 64 bit binaries.
- **Solution 2**: Write code that works with both binaries.

Automatically pad to 64 bit

```
typedef union
{
    struct
    {
        unsigned int h;
        unsigned int l;
    };
    unsigned long long e;

    struct
    {
#ifndef __powerpc64__
        unsigned int reserved;
#endif
        void *p;
    };
} ea_t;
```

```
/* On PPE */
char buf[4096];

ea_t bufptr;
bufptr.p = buf;


/* When issuing DMA commands
   on SPE use .h and .l as
   arguments to the 64 bit
   version */

spu_mfcdma64( mybuf,
   bufptr.h, bufptr.l,
   4096, tag, MFC_GET_CMD );
```

Useful gcc directives

Align to boundary (PPE and SPE)

```
typedef struct foo
{
        char stuff[5];
} __attribute((aligned(16))) foo_t;
```

sizeof(foo_t) % 16 == 0 in addition to 16 B pointer alignment!

Static branch hinting (SPE)

```
if(__builtin_expect(foo > 10, 0))
{
        /* Condition unlikely. We expect foo to be <= 10 */
}
```

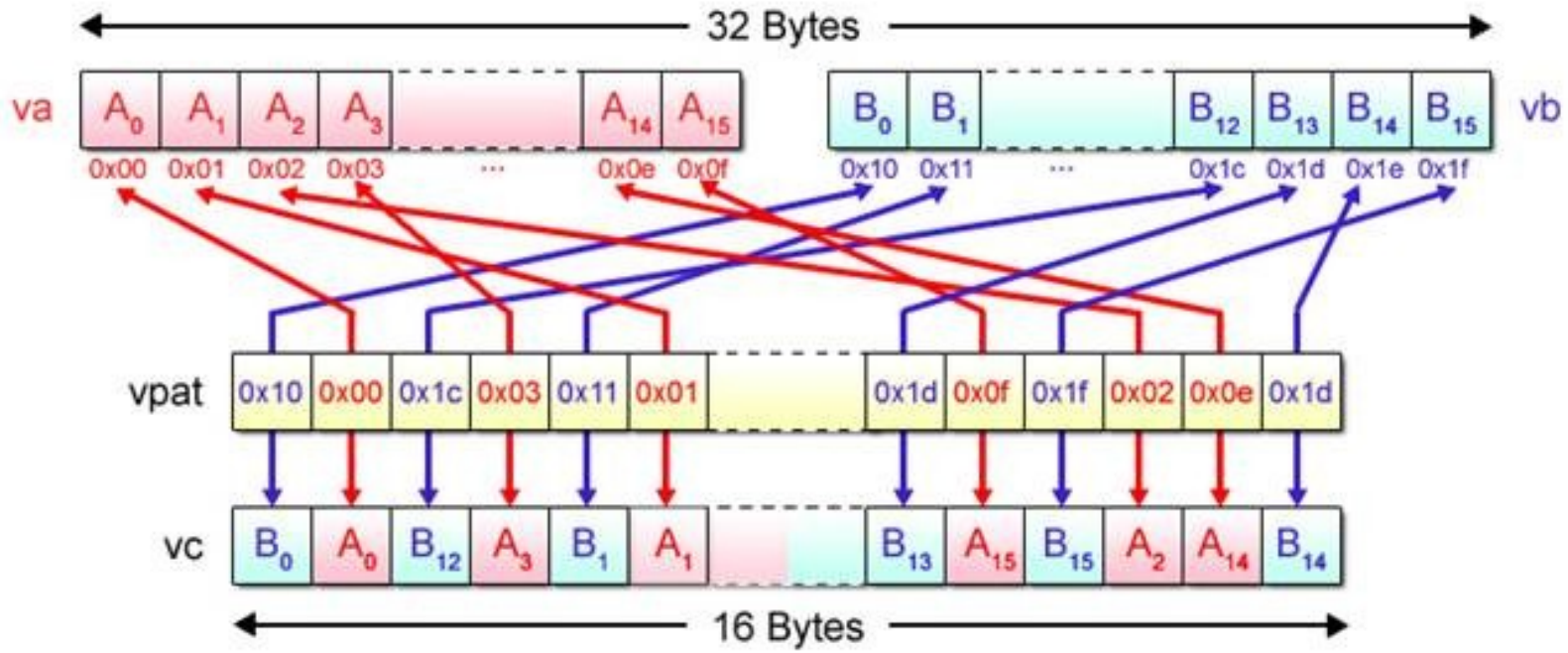
Prefetch data to reduce cache miss-latency (PPE)

```
for (i = 0; i < n; i++)
{
        a[i] = a[i] + b[i];
        __builtin_prefetch(&a[i+j], 1, 1);
        __builtin_prefetch(&b[i+j], 0, 1);
        /* ... */
}
```

SIMD Programming Techniques

- Thinking SIMD is hard and requires that the programmer really understands the data structures.
- When correctly applied, SIMD can result in dramatic increase in throughput.
- Intrinsics support in the gcc compiler, so writing assembly is (usually) not necessary.

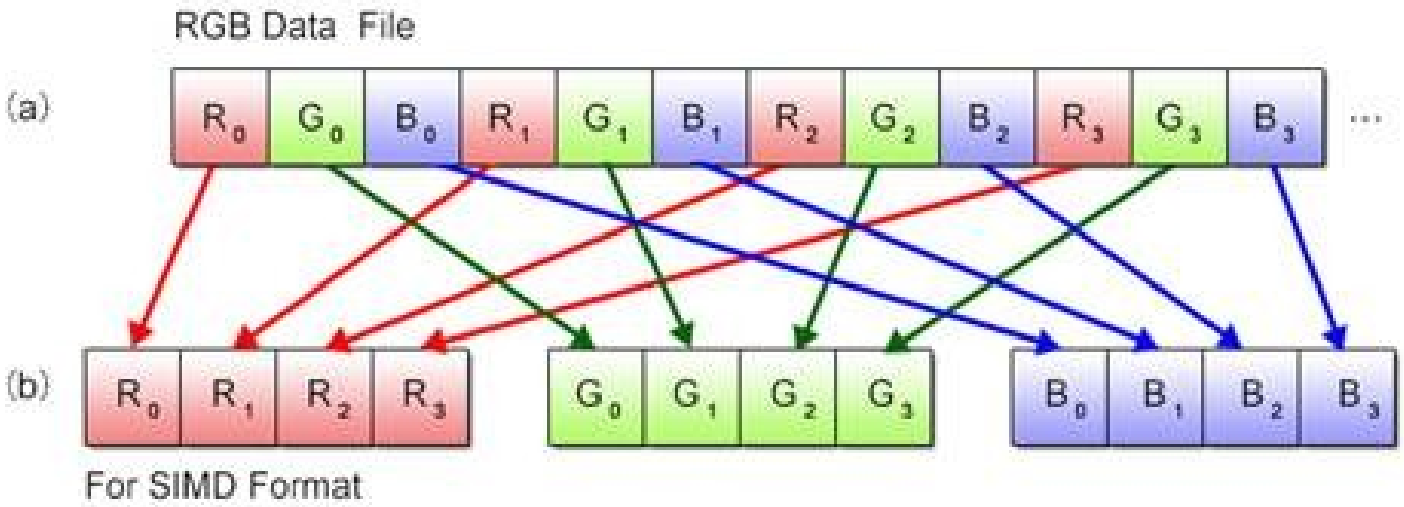
Vector permutation



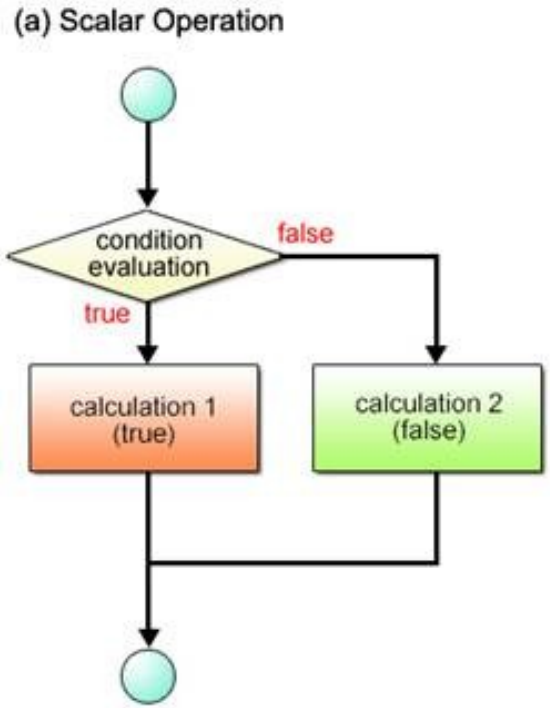

```
PPE: vc = vec_perm(va, vb, vpat);

SPE: vc = spu_shuffle(va, vb, vpat);
```

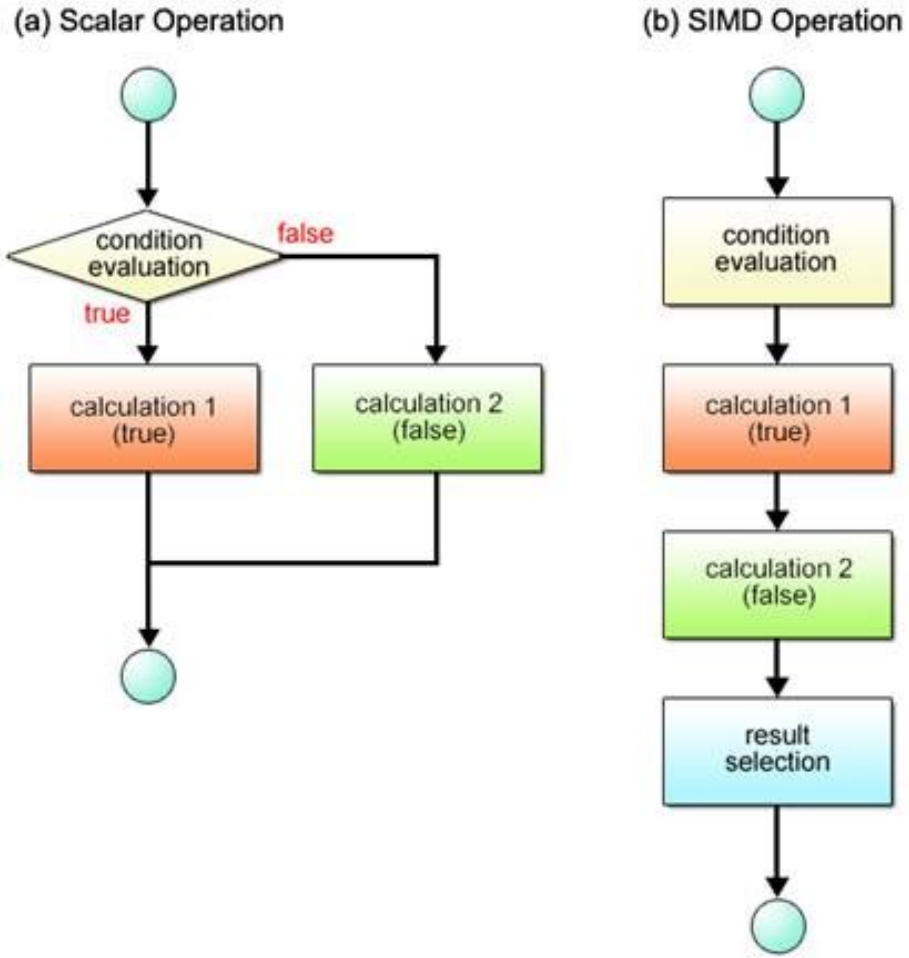
Vector Permutation Example



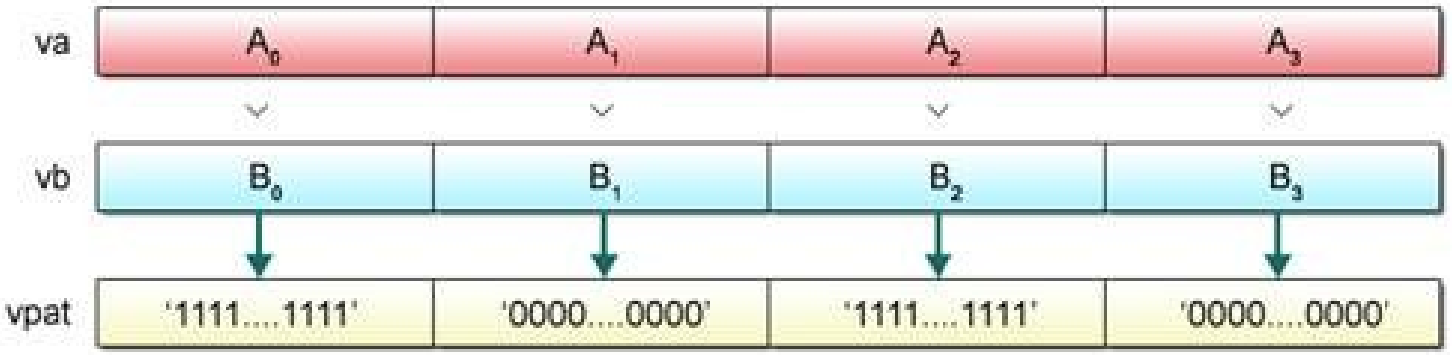
Branch-free SIMD processing



Branch-free SIMD processing



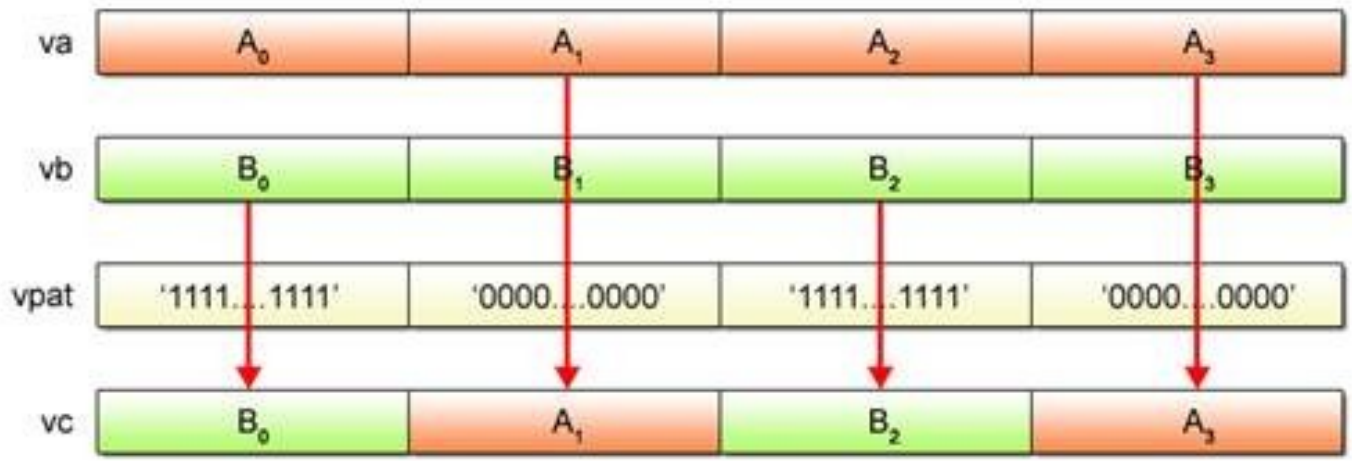
Vector Comparison




```
PPE: vpat = vec_cmpgt(va, vb);

SPE: vpat = spu_cmpgt(va, vb);
```

Vector Selection




```
PPE: vc = vec_sel(va, vb, vpat)

SPE: vc = spu_sel(va, vb, vpat)
```

Image Gray scaler Example

- A simple program that extracts the luminance component of an input image.
- Detailed design and source code available in the *Cell Programming Primer.*


```
f_Y(R, G, B)  R x 0.29891 + G x 0.58661 + B x 0.11448
```

Image Gray scaler Example

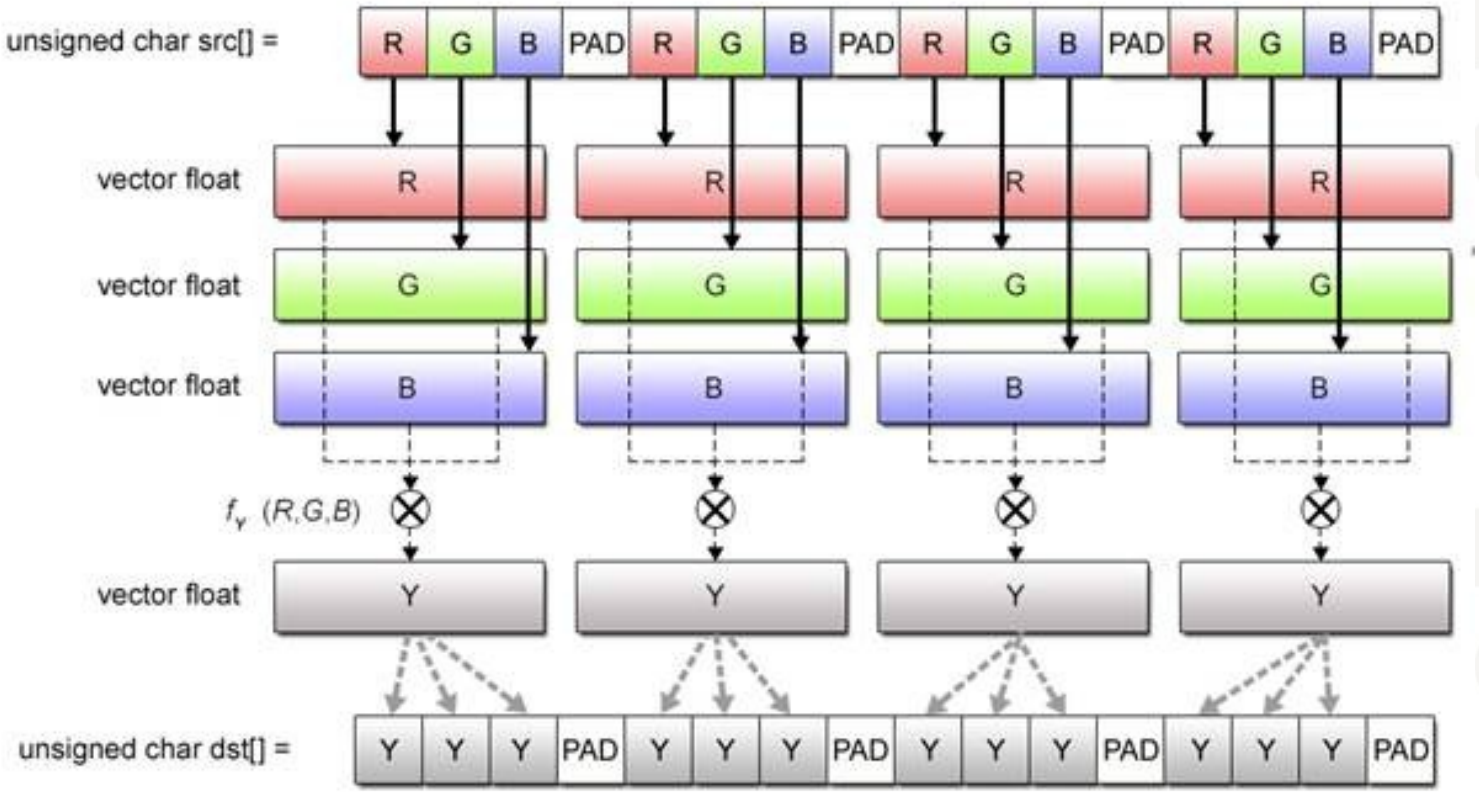


Image Gray scaler Example

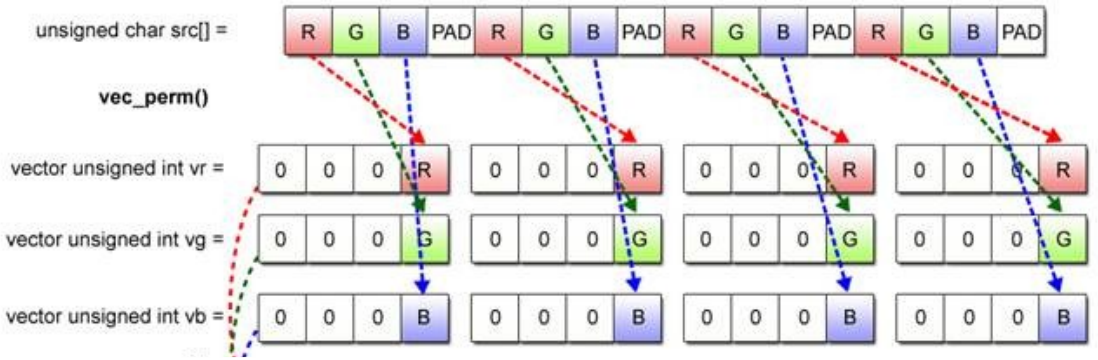


Image Gray scaler Example

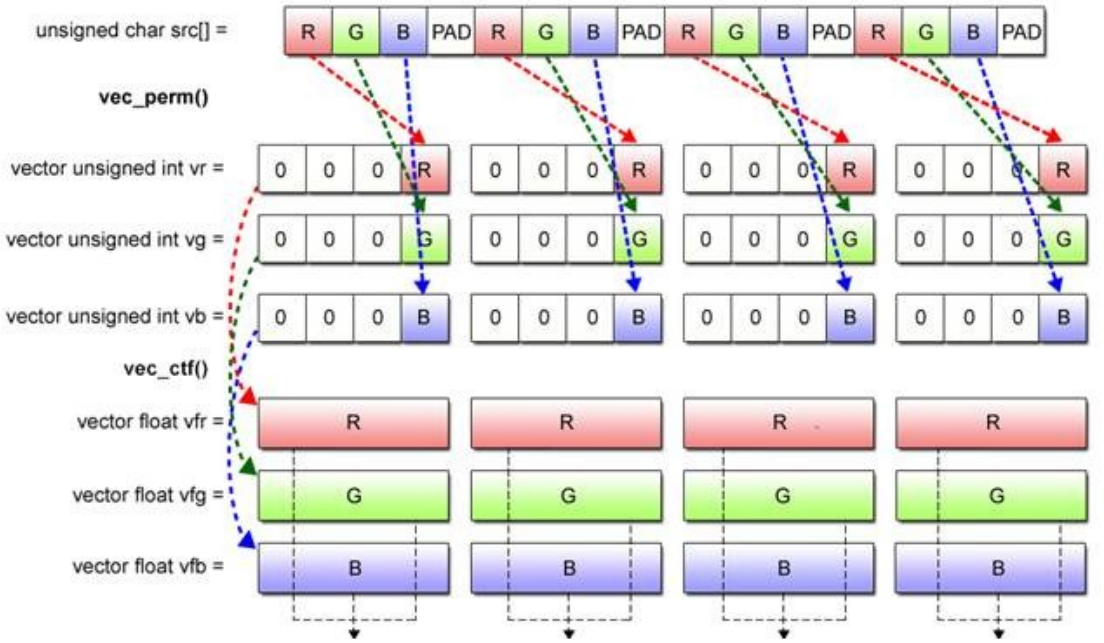


Image Gray scaler Example

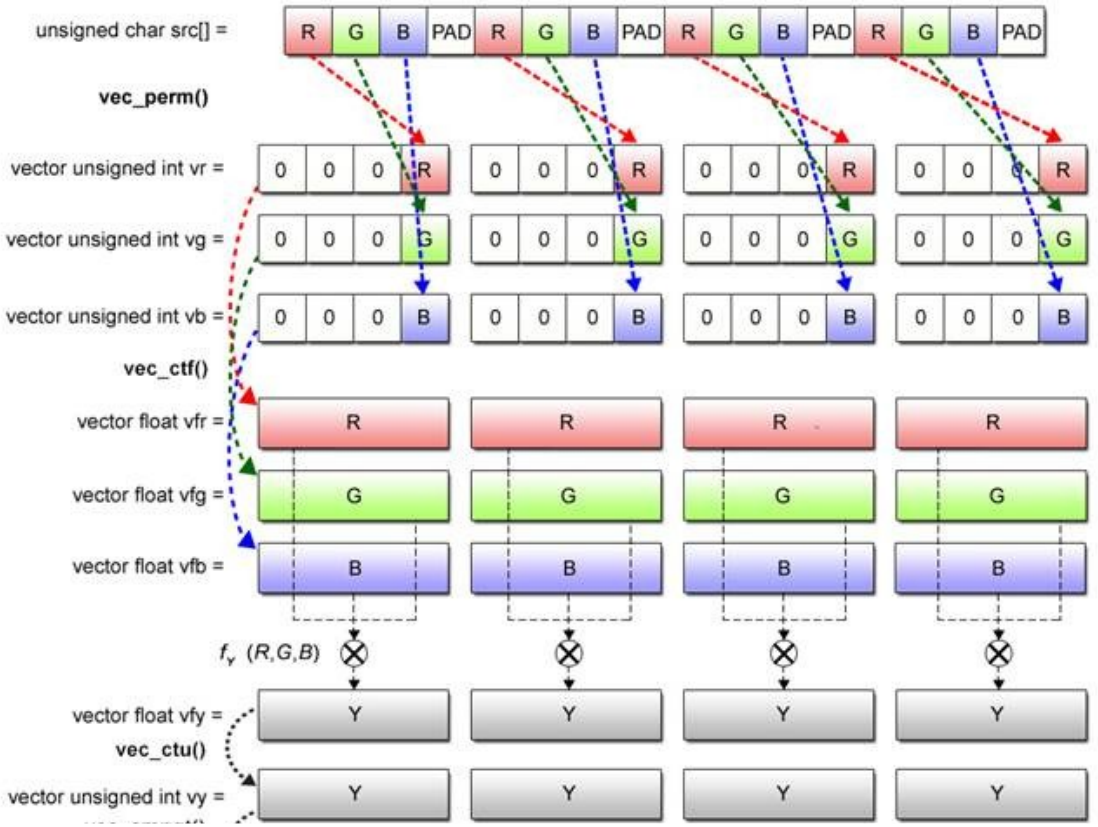
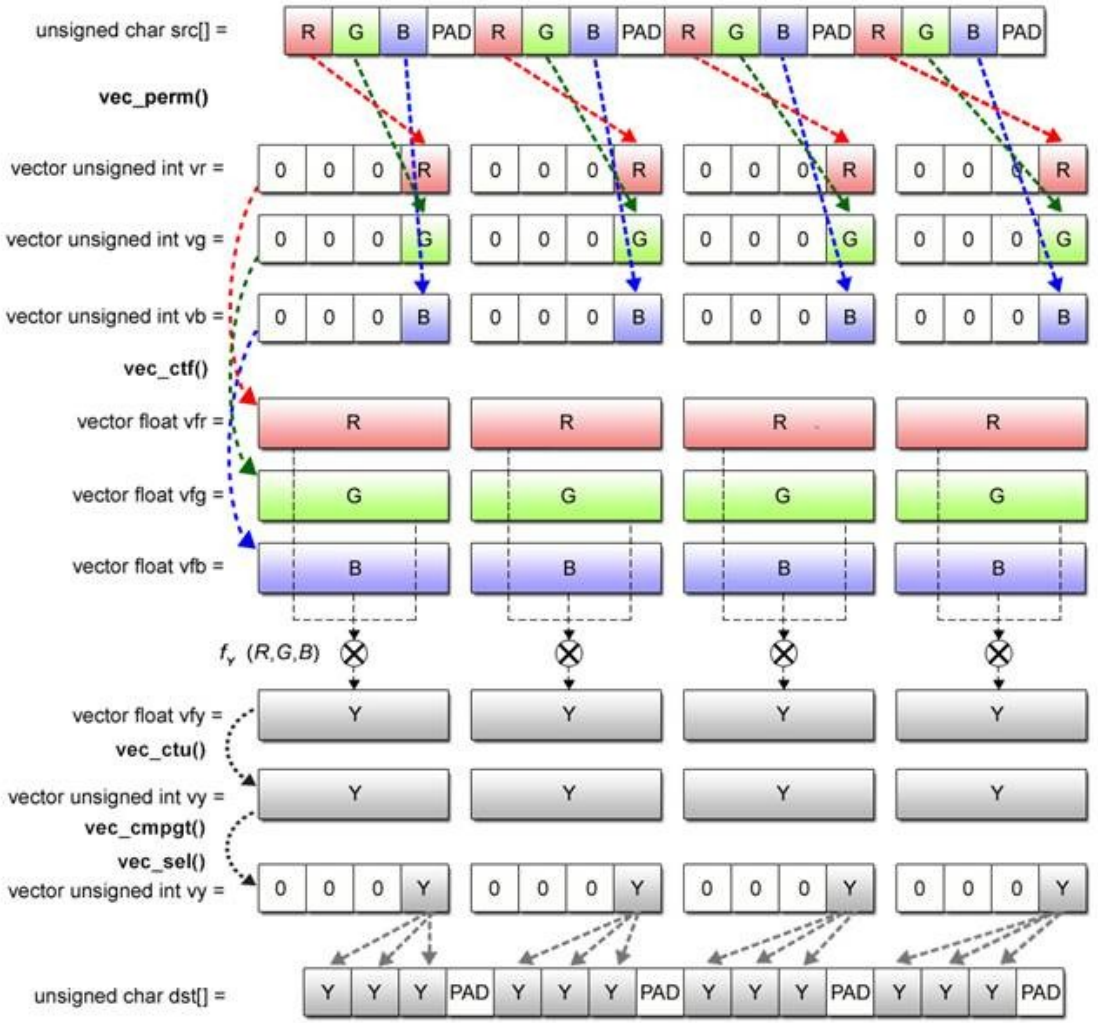


Image Gray scaler Example



Steps for parallelizing a program

- Step 1: Understanding the problem:
  - **Performance objectives**
    - Is performance critical?
  - **Type of computation**
    - PPE or SPE better suited for this type of work?
  - **Data types**
    - Fixed point? Float Point? Single/Double? Byte/word?
  - **Data access patterns**
    - Random or sequential access?

Step 2:Programming tools and tech.

- System. Conf / availability
- Quality and reliability
- Programmer skills
- Schedule objectives
- Performance objectives

Step 3: High-level parall. strategy

- Assign work to SPEs:
  - **Algorithmically**: If w work and n SPEs, assign w/n work to every SPE.
  - **Self-managed**: Make SPEs arbitrate for work to provide load balancing.
  - **Mastered**: One processor distributed work to the others using a work queue in main memory or by using mailboxes or signals.

Step 4: Low level strategy

- Optimize individual parts to the target arch.:
  - Use intrinsics (SIMD)
  - Alternatively, OpenMP #pragma
  - Identify data structures for vector instructions

Step 5: Design effective data struct.

- Organize structurs and arrays to SIMD format:
  - Consider padding data to an even multiple of quadwords to avoid expensive bitshifts before processing.
  - Align to quadwords.
  - Consider cache effects on shared data and synchronization variables.
  - Optimize data structure layout.

Step 6: Iterate and refine

- Determime if current parallelization strategy is effective.
- If not, revise and go to step 3.
- A profiler would be nice here... Sorry!

Step 7: Fine tune

- Avoid branches.
- Inline frequently called functions.
- Unroll loops.
- Remove false dependencies.
- Consider memory bank accesses.
- Remove inefficient operations and instructions.

Summary

- Adapt your program to exploit the target hardware's capabilities, and avoid its limitations. Know your architecture!
- Consider the memory flow of your program.
- Having multiple cores is pointless if they are idle. Make sure they always have work to do!

Useful Resources

Cell Broadband Engine Programming Handbook (IBM)
https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D/$file/CellBE_PXCell_Handbook_v1.11_12May08_pub.pdf

Cell Programming Primer (Sony)
http://cell.fixstars.com/ps3linux/download/cell-linux-20080201/CELL-Linux-CL_20080201-ADDON/doc/CellProgrammingPrimer.html

Sony Documentation for Cell Broadband Engine™ Architecture
http://cell.scei.co.jp

C/C++ Language Extensions for Cell Broadband Engine™ Architecture (Sony)
http://cell.scei.co.jp/pdf/Language_Extensions_for_CBEA_v23.pdf

SPE Runtime Management Library
http://moss.csc.ncsu.edu/~mueller/cluster/ps3/doc/libspe-v2.0.pdf