Playing with parallelism on PS3s

W. P. Petersen

Seminar for Applied Mathematics, ETH Zürich
http://www.math.ethz.ch/∼wpp/
e-mail: wpp@math.ethz.ch

28 March, 2007, LBL

Justification for PS-3 experiments:

- Price/performance: PS-3 costs ∼ \$500/unit with 6-7 working SPEs; cheap compared to say an Opteron 244.
- more levels of parallelism:
  * multiple independent functional units,
  * instruction (pipelines) + vectors,
  * independent threads (6-7 SPEs),
  * message passing between CELLs.
- Heat: Itanium generates ∼130 watts, while CELL is only 40-60. The whole Playstation takes ∼ 180 – 200.

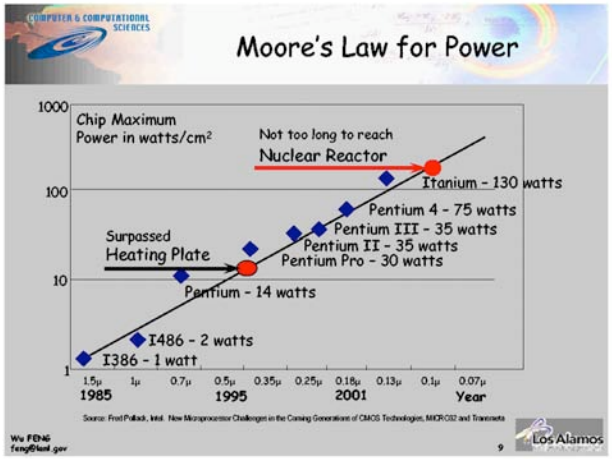

Figure: Microprocessor power density: doubles ∼ 2 years.

Current PS-3 machines are poorly suited for scientific/engineering simulations:

- memories are too small (512 MB with 1/2 of this usable) and no substitution,
- programming in threads remains painful (Open MP ?),
- single precision arithmetic is inadequate, double is OK,
- Giga-bit Ethernet PCI switches are not easily replaced,
- MPI using Ethernet is too green.

PS3 has all the architectural features, however.

- two important components:
  * PPE is a conventional PowerPC
  * the eight SPEs are simplified PowerPCs with 128 Altivec registers per SPE, ring network is tight,
- global clock is 3.2 GHz,
- DRAM to memory unit has 25.6 GB/s BW (128 bit wide bus),
- More expensive versions (e.g. Mercury) have 2 CELLs/blade,
- vector alignment is in local memory.

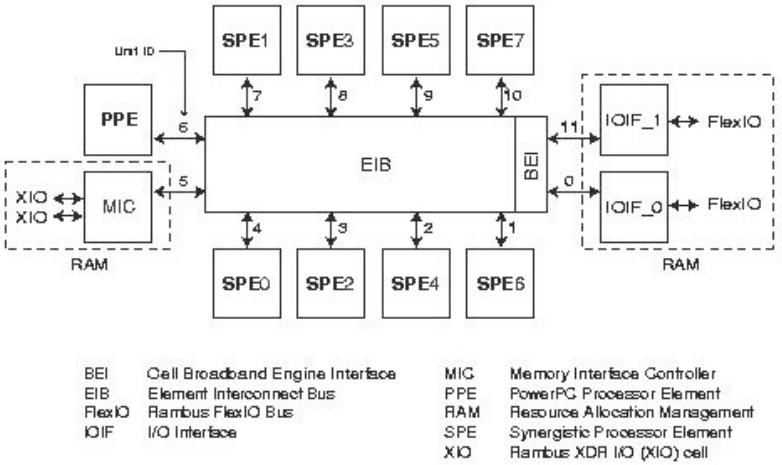

Figure: CELL memory control (CBE programming handbook)

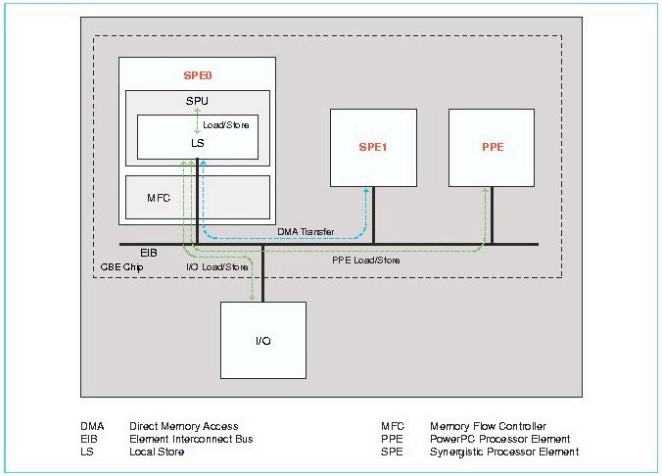

Figure: Memory access methods (CBE programming handbook)

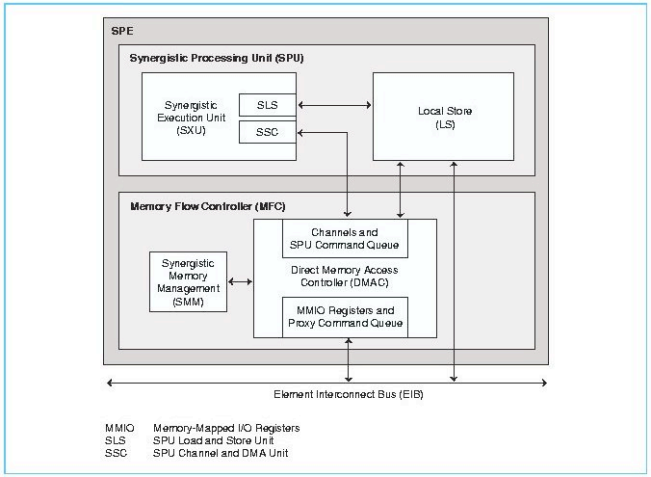

Figure: SPE memory flow control (CBE programming handbook)

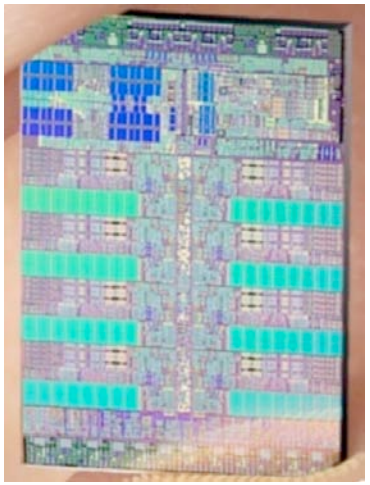

Figure: Basic CELL processor held between thumb and forefinger

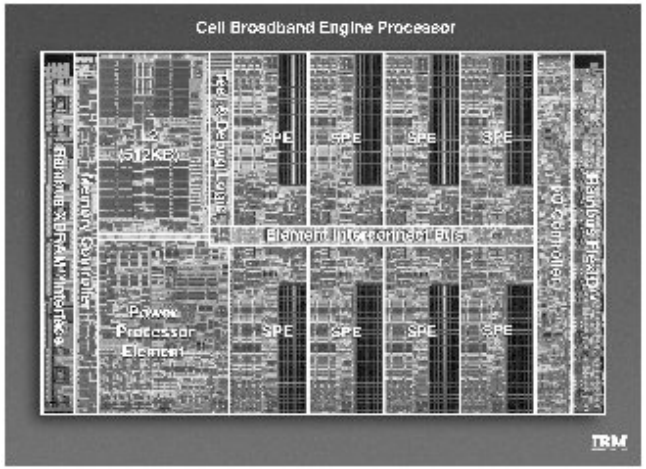

Figure: Realestate in CELL

The crew:

- ETH students:

  Denis Nordmann, François Gaignat, Stephan Gammeter, Lukas Gamper, Patrick Mächler, Robin Krom, Kaspar Müller, Mauro Calderara, Christine Tobler, Bastian Pentenrieder
- ETH staff:

  Professors: wpp, Rolf Jeltsch, Jörg Waldvogel (Math), Matthias Troyer (Physik), Anton Gunzinger (EE).
- University of Zürich students:

  Jonathan Coles, Alessandro Viretta.
- Univ. of Zürich staff:

  Prof. George Lake (CSE/Astrophysik).

Figure: Informal team from ETH and Univ. ZH. Left to right: Coles, Krom, wpp, Waldvogel, Pentenrieder, Gaignat, Lake, Tobler, Nordmann, and Müller. Our first PS3 is on Gaignat and Lake's knees.


Figure: Matthias Troyer, Lukas Gamper, and Rolf Jeltsch

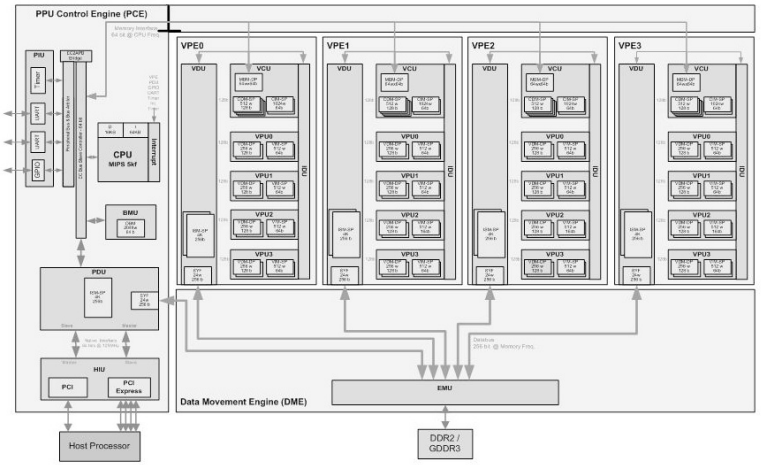

Figure: Alternative video games processor: AGEIA

Levels of parallelism:

- Instructions are pipelined. Out-of-order and branch prediction are available only on PPE. SPE executes in linear order with $\sim$20 cycle prediction miss penalty. Typical execution times are 2-7 cycles.

```
if(e(x)){
     y = f(x);
} else {
     y = g(x);
}
```

- Pipelines and vectorization. SPE units each have 128 **vector registers** of four 32-bit words each; that is, 128 sixteen byte vectors.
- Automatic vectorization remains difficult due to block alignment. **However**, 128 vector registers compares favorably to only 32 on PowerPC.
- *Intrinsics* remain the principal vectorization paradigm: unrolling in groups of 4 (single) or 2 (double) to use these intrinsics.

An example: compute π by inscribing a unit circle inside a *side* $= 2$ square. The area of an inscribed quadrant is $\pi/4$. PPE part:

```
#include <stdlib.h> /* Modified IBM/Sony/Toshiba code */
#include <stdio.h>
#include <errno.h>
#include <libspe.h>
#include <math.h>
#include <iostream>
#include "context.h"
extern spe_program_handle_t bench_spe;
#define SPE_THREADS 6
const int iterations=6*10000000;
int main(int argc, char* argv[])
{
  simulate_on_spe(); return (0);
}
```

PPE part(cont.)

```
void simulate_on_spe(){
 speid_t spe_ids[SPE_THREADS];
 context ctxs[SPE_THREADS] __attribute__ ((aligned(16)));
 char* memory[SPE_THREADS];
 int hits=0,i,status;
 for(i=0;i<SPE_THREADS;i++){
    ctxs[i].iterations = iterations/SPE_THREADS;
    ctxs[i].hits =
       (int*)malloc_aligned(16,sizeof(int),memory[i]);
    spe_ids[i] =
     spe_create_thread(0,&bench_spe,&ctxs[i],NULL,-1,0);
    if(spe_ids[i] == 0) exit(1);
 }
```

PPE part (cont.)

```
/* Wait for SPE-thread to complete execution.*/
  for(i=0; i<SPE_THREADS; i++) {
     (void)spe_wait(spe_ids[i], &status, 0);
  }
  for(i=0;i<SPE_THREADS;i++) {
     hits += *ctxs[i].hits; free(memory[i]);
  }
  cout << "Pi = "<< 4.0*hits/iterations
}
```

SPE portion

```
#include <math.h> /* Modified IBM/Sony/Toshiba code */
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "../context.h"
volatile context ctx;
int main(unsigned long long spu_id,
         unsigned long long parm)
{
  unsigned int init[16],tag_id = 0;
  int i,hits,InitWELLRNG512a(unsigned int*);
  double sd,ggl(double*);
  spu_writech(MFC_WrTagMask, -1);
  spu_mfcdma32((void *)(&ctx),(unsigned int)parm,
               sizeof(context),tag_id,MFC_GET_CMD);
  (void)spu_mfcstat(2); sd = (double) spu_id;
```

SPE portion (cont.)

```
  for(i=0;i<16;i++) init[i]=(unsigned int)5.0e+9*ggl(&sd);
  InitWELLRNG512a(init);
  simulate(ctx.iterations,&hits);
  spu_mfcdma32((void *)(&hits), (unsigned int)(ctx.hits),
               sizeof(int), tag_id, MFC_PUT_CMD);
  (void)spu_mfcstat(2); return (0);
}
```

One other piece: `context.h`

```
typedef struct {
        int iterations;
        int *hits;
        int dummy1;
        int dummy2;
} context;
```

Actual calculation:

```
void simulate(int n, int* hits){
   int i,hits_count=0;
   double WELLRNG512a();
   double x,y;
   for(i=0;i<n;i++){
      x = WELLRNG512a(); y = WELLRNG512a();
      if ((x*x+y*y) < 1.0) hits_count++;
   }
   *hits=hits_count;
   return;
}
```

Compilers are `ppuxlc++` amd `spuxlc`

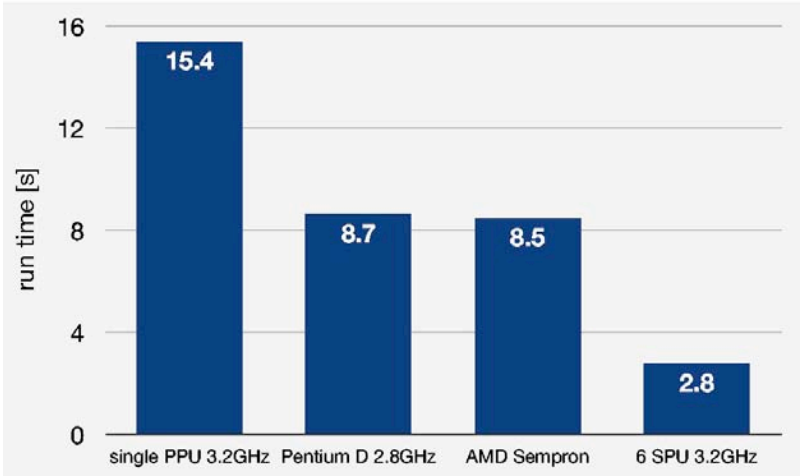

Figure: Six SPE results: **No vectorization**. Stephan Gammeter data using IBM code and `srand48`.

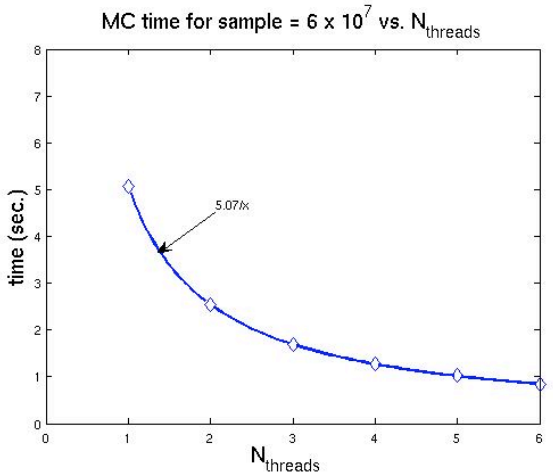

Figure: MC comp. time for π with $N_{sample} = 6 \times 10^7$ vs. $N_{threads}$. Speedup=5.9. wpp data using `WELLRNG512a`.

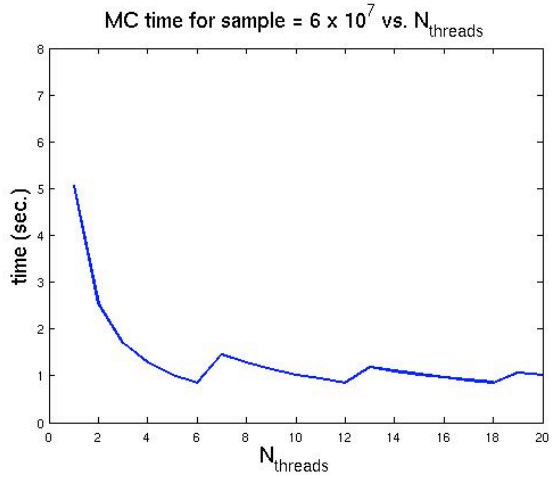

Figure: MC comp. time for π with more threads: used `WELLRNG512a`.

Comments on parallel Monte-Carlo:

- Usually almost embarrassingly parallel - each sample path is independent. Only functionals require communication.
- Independent random number streams are easily arranged. Typical RNGs are of form:

$$x_n = x_{n-p} \ \mathbf{op} \ x_{n-q}$$

and are seeded with a buffer bigger than $p \vee q$ initial values $\mathbf{x}$.

L'Ecuyer and Panneton's `WELLRNG512a` uses a circular buffer containing sixteen integers. I used `ggl` seeded with the process ID to initialize this buffer.

In looking for dense clusters of primes, an interval $[a, b]$ is split into $\sim (b-a)/(2 \cdot N_{threads})$ odd numbers to test for primality: these are loaded in groups of `dma_length`.

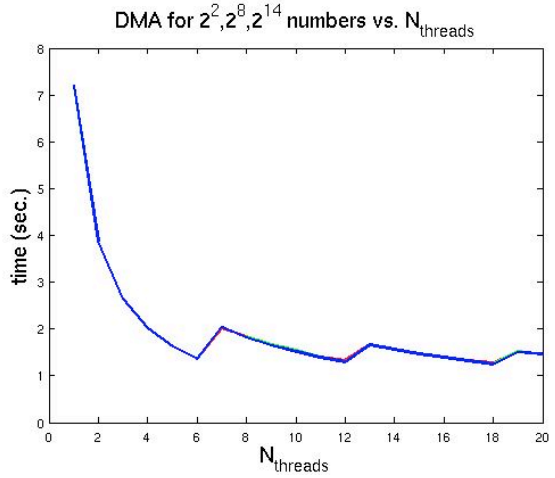

Figure: Direct DMA transfer vs. $H_{threads}$. François Gaignat data.

Problems for vectorization:

- Little/no autovectorization `xlc` - maybe `gcc4.4`?
- use of Altivec intrinsics
- data alignment
- double data? In that case, $VL = 2$.

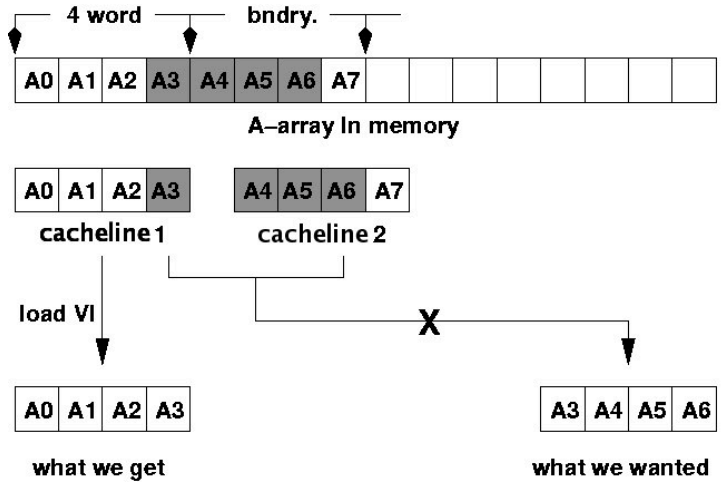

Figure: Short vectors and block alignment.

Two forms of vectorization: long VL, and short VL.

$$\begin{aligned} VL &= \text{vector length, no. of indep. data} \\ PL &= \text{pipeline length} \\ \text{speed-up} &= \text{scalar/vector timing ratio} \end{aligned}$$

Scalar mode takes $PL \cdot VL$ cycles, but vector only $VL + PL$, so

$$\text{speed-up} = \frac{PL \cdot VL}{VL + PL}$$

When VL large compared to PL, (Cray, Fujitsu, NEC, Hitachi)

$$\text{speedup} \approx PL$$

But if PL large (memory pipe dominates: Pentiums, PowerPC),

$$\text{speedup} \approx VL$$

An example: self-sorting FFT

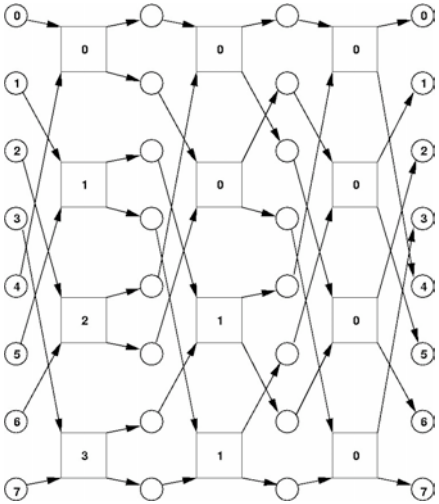

Figure: Workspace version of self-sorting FFT

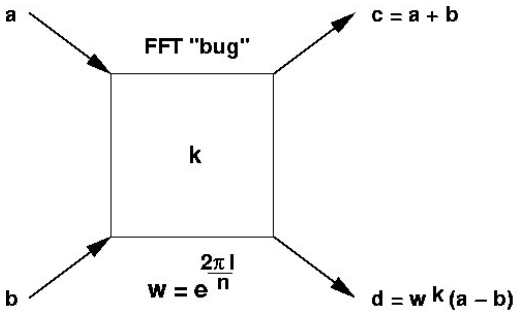

Figure: Decimation in time computational FFT "bug"

$$V_3 = \begin{bmatrix} (a-b)_r \\ (a-b)_i \\ (a-b)_r \\ (a-b)_i \end{bmatrix}, \quad V_6 = \begin{bmatrix} \omega_r \\ \omega_r \\ \omega_r \\ \omega_r \end{bmatrix}, \quad V_7 = \begin{bmatrix} -\omega_i \\ \omega_i \\ -\omega_i \\ \omega_i \end{bmatrix},$$

$$V_3 \xrightarrow{shuffle} V_4 = \begin{bmatrix} (a-b)_i \\ (a-b)_r \\ (a-b)_i \\ (a-b)_r \end{bmatrix}, \quad V_0 = V_6 \cdot V_3 = \begin{bmatrix} \omega_r(a-b)_r \\ \omega_r(a-b)_i \\ \omega_r(a-b)_r \\ \omega_r(a-b)_i \end{bmatrix},$$

$$V_1 = V_7 \cdot V_4 = \begin{bmatrix} -\omega_i(a-b)_i \\ \omega_i(a-b)_r \\ -\omega_i(a-b)_i \\ \omega_i(a-b)_r \end{bmatrix}, \quad \mathbf{d} = V_2 = V_0 + V_1. \quad (result)$$

Figure: Complex arithmetic for $\mathbf{d} = w^k(\mathbf{a} - \mathbf{b})$ using Altivec. $\mathbf{c} = \mathbf{a} + \mathbf{b}$ is easy.

My standard FFT benchmark (`ppuxlc -O3`):

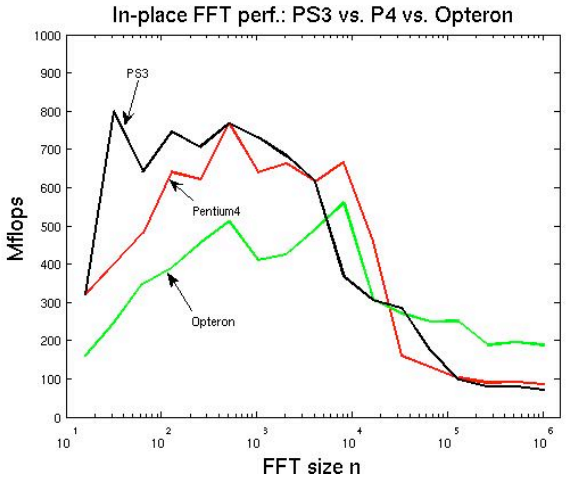

Figure: In-place FFT on PS-3 PPE vs. Pentium-4 and Opteron 242.

Workspace version of FFT (used `ppuxlc -O3 -qaltivec`, and alignment via `valloc`):

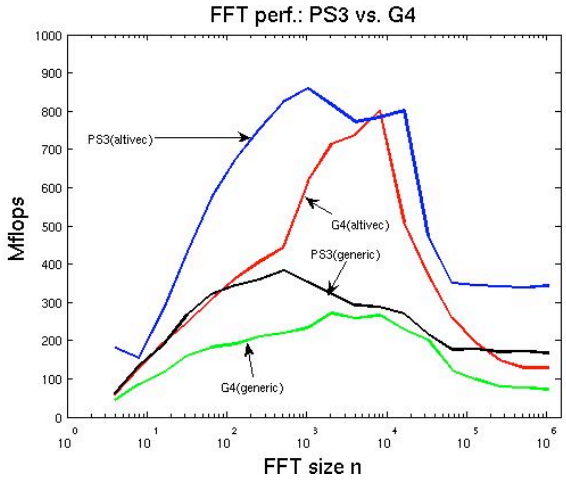

Figure: PS-3 PPE vs. G-4 (Ogdoad), generic vs. Altivec.

With much loop unrolling and twiddle-factor expansion.

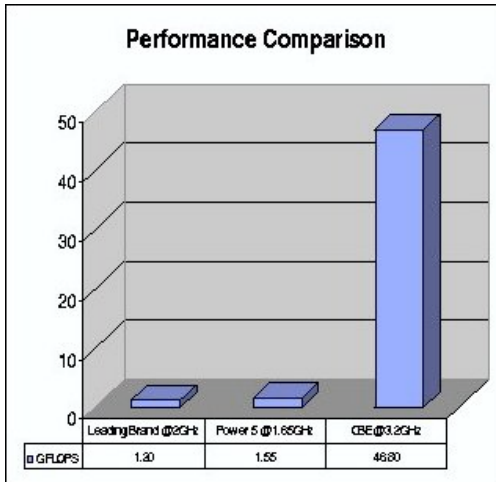

Figure: Chunghen Chow et al, Large (2^{24}) FFT report data.

Multiple PS3 experiments: PCI switches are for Giga-bit Ethernet, but unchangeable

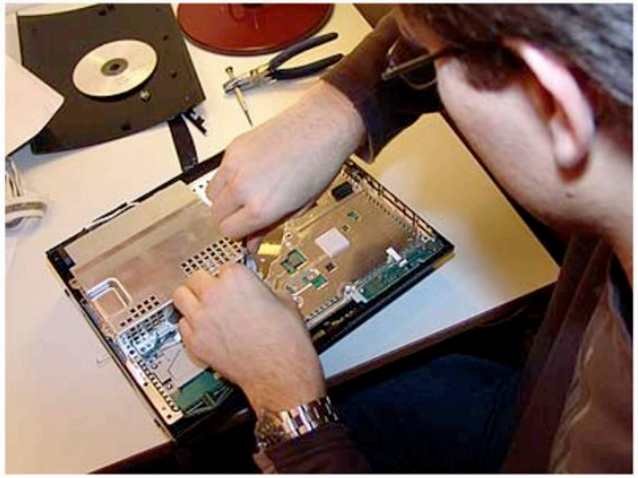

Figure: Inside a PS3: roundtrip message passing $\geq 220\mu$S

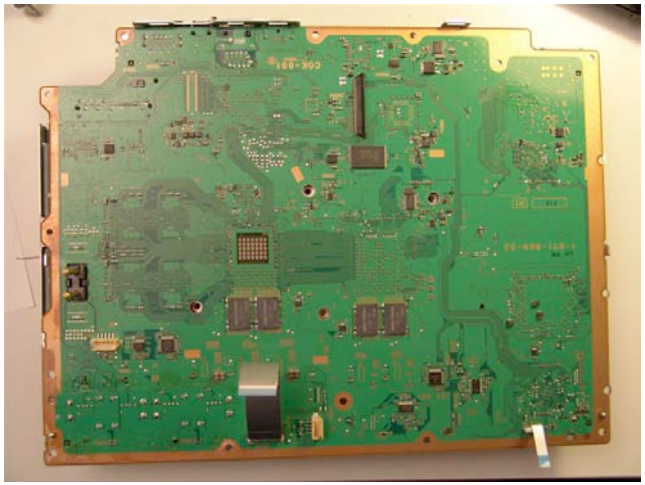

Figure: PS3 circuit board: backside shows CELL mount.

Conclusions and outlook:

- CELL Broadband engine, in its cheap form (Playstation-3), has all the forms of parallelism we know.
- As an educational device, PS3s allow study both of hardware and algorithms.
- Even using only the control processor (PPE), a PS3 is a powerful Linux machine.
- Even for folks with no interest in games, they're fun to play with.

References:

- `http://cluster.qubits.ch/`
- Alex Chow, G.C. Fossum, and D.A. Brokenshire, *A Programming Example: Large FFT on the Cell Broadband Engine*, IBM Technical Report, 2005.
- IBM *Cell Broadband Engine Programming Tutorial*, vers. 2.0, Dec. 15, 2006.
- `http://www-128.ibm.com/developerworks/power/cell/docs_articles.html`
- Sam Williams, *Structured Grids and Sparse Matrix-Vector Multiplication on the Cell Processor*, LBL talk, Nov. 1, 2006.
- IBM *CELL Broadband Engine Programming Handbook*, version 1, April 19, 2006.