Optimal thread-level parallelism on Cell and x86-based multi-core architectures using the same code: A case study using elastic finite-difference modeling.

Samuel Brown
University of Utah

March 4, 2008

Motivation

1. Compute synthetic data quickly on available hardware.

Motivation

1. Compute synthetic data quickly on available hardware.
2. Exclusive access to PS3.

Motivation

1. Compute synthetic data quickly on available hardware.
2. Exclusive access to PS3. (hmm?)

Motivation

1. Compute synthetic data quickly on available hardware.
2. Exclusive access to PS3. (hmm?)
3. Identify the common ground shared by homogeneous and heterogeneous multi-core processors.

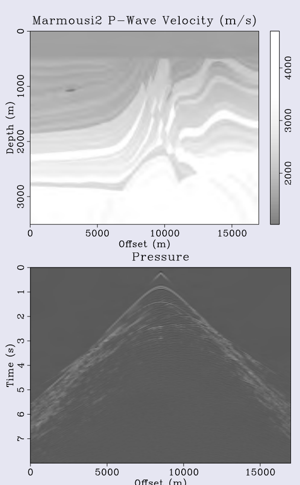
Motivation

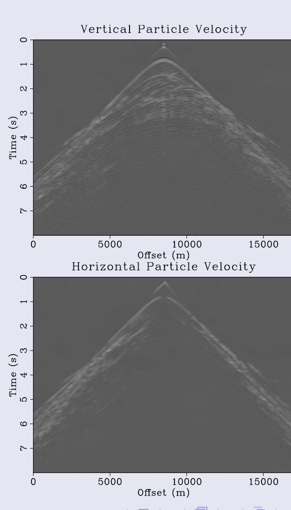
1. Compute synthetic data quickly on available hardware.
2. Exclusive access to PS3. (hmm?)
3. Identify the common ground shared by homogeneous and heterogeneous multi-core processors.
4. Discover abstractions to create more common ground.

Motivation

1. Compute synthetic data quickly on available hardware.
2. Exclusive access to PS3. (hmm?)
3. Identify the common ground shared by homogeneous and heterogeneous multi-core processors.
4. Discover abstractions to create more common ground.
5. Maintain performance.

Marmousi2 Seismograms





Coupled First Order PDEs

$$\rho \frac{\partial v_x}{\partial t} = \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xz}}{\partial z}$$

$$\rho \frac{\partial v_z}{\partial t} = \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{zz}}{\partial z}$$

Coupled First Order PDEs

$$\rho \frac{\partial v_x}{\partial t} = \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xz}}{\partial z}$$

$$\rho \frac{\partial v_z}{\partial t} = \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{zz}}{\partial z}$$

$$\frac{\partial \tau_{xx}}{\partial t} = (\lambda + 2\mu) \frac{\partial v_x}{\partial x} + \lambda \frac{\partial v_z}{\partial z}$$

$$\frac{\partial \tau_{zz}}{\partial t} = (\lambda + 2\mu) \frac{\partial v_z}{\partial z} + \lambda \frac{\partial v_x}{\partial x}$$

$$\frac{\partial \tau_{xz}}{\partial t} = \mu \left(\frac{\partial v_x}{\partial z} + \frac{\partial v_z}{\partial x} \right)$$

Time Marching

4th order in space, 2nd order in time Madariaga-Virieux staggered grid scheme [Levander, 1988].

1. Compute velocity fields, v_x and v_z, at current timestep.
2. Write velocity components of seismograms to file.
3. Add pressure sources.
4. Compute stress fields, τ_{xx}, τ_{zz}, and τ_{xz}, at current timestep.
5. Write pressure component of seismograms to file.

Cell Architectural Challenges/Advantages

1. 1 general purpose core + 6-8 specialized cores (SPEs)

Cell Architectural Challenges/Advantages

1. 1 general purpose core + 6-8 specialized cores (SPEs)
2. SPEs have 256 KB local memory

Cell Architectural Challenges/Advantages

1. 1 general purpose core + 6-8 specialized cores (SPEs)
2. SPEs have 256 KB local memory
3. Data must be explicitly moved between local memory and main memory through DMA

Cell Architectural Challenges/Advantages

1. 1 general purpose core + 6-8 specialized cores (SPEs)
2. SPEs have 256 KB local memory
3. Data must be explicitly moved between local memory and main memory through DMA
4. Local SPE memory is like a user-controlled cache

Cell Architectural Challenges/Advantages

1. 1 general purpose core + 6-8 specialized cores (SPEs)
2. SPEs have 256 KB local memory
3. Data must be explicitly moved between local memory and main memory through DMA
4. Local SPE memory is like a user-controlled cache
5. Out-of-core methodology: main memory to local memory

Control vs. Compute

Control thread handles:

- Initialization
- All file I/O
- Thread synchronization

Control vs. Compute

Control thread handles:

- Initialization
- All file I/O
- Thread synchronization

Compute threads handle:

- Computational kernel

Control vs. Compute

Control thread handles:

- Initialization
- All file I/O
- Thread synchronization

Compute threads handle:

- Computational kernel
- That's about it

SIMD

- Grids are partitioned along the z axis into four equal parts

SIMD

- Grids are partitioned along the z axis into four equal parts
- The four partitions are then interleaved in memory

SIMD

- Grids are partitioned along the z axis into four equal parts
- The four partitions are then interleaved in memory
- Scalar finite-difference code is altered by replacing arithmetic operators with intrinsics

Intrinsics

```
#ifdef CELL

#include <spu_intrinsics.h>

#define VECFLOAT vector float

#define ADDV(a,b)    spu_add(a,b)
#define SUBV(a,b)    spu_sub(a,b)
#define MULV(a,b)    spu_mul(a,b)
#define MADDV(a,b,c) spu_madd(a,b,c)

#else

#include <xmmintrin.h>

#define VECFLOAT     __v4sf

#define ADDV(a,b)    __builtin_ia32_addps(a,b)
#define SUBV(a,b)    __builtin_ia32_subps(a,b)
#define MULV(a,b)    __builtin_ia32_mulps(a,b)
#define MADDV(a,b,c) __builtin_ia32_addps(__builtin_ia32_mulps(a,b),c)

#endif
```

Control Loop

```
for (it=0; it<nt; it++) {
   //--- velocity updates
   thread_sync_control(wd, 2*it);
   update_ghosts_velocity(vx, vz);
   thread_wake_control(wd, 2*it);
   //--- write receivers
   if(it == i_recv_write && c_recv_write < ntr) {
      write_recv2d(vx, recv_buf, recv_vx, nrecv, fd_vx, 4);
      write_recv2d(vz, recv_buf, recv_vz, nrecv, fd_vz, 4);
   }
   //--- stress updates
   thread_sync_control(wd, 2*it+1);
   update_ghosts_stress(txx, tzz, txz);
   //--- add sources
   if(it<ntsrc) {
      add_src2d(txx, src, srcsig, nsrc, it, 4);
      add_src2d(tzz, src, srcsig, nsrc, it, 4);
   }
   thread_wake_control(wd, 2*it+1);
   //--- write receivers
   if(it == i_recv_write && c_recv_write < ntr) {
      write_press2d(txx, tzz, recv_buf, recv_p, nrecv, fd_p, 4);
      c_recv_write++;
      i_recv_write = get_next_write_time(c_recv_write, axt, axrt);
   }
}
```

Compute Loop

```
#ifdef CELL
int main(unsigned long long spe_id,
         unsigned long long parm) {
   struct tdata *td;
   struct pdata *pd;
   int it;
   spe_init(&td, &pd, parm);
#else
void *thread_compute(void *arg) {
   struct tdata *td = (struct tdata*)arg;
   struct pdata *pd = (struct pdata*)td->pd;
   int it;
   pthread_mutex_lock(td->mutex);
#endif

   //--- timesteps
   for (it=0; it<pd->nt; it++) {
      update_velocity(pd);
      thread_sync_compute(td, 2*it);
      update_stress(pd);
      thread_sync_compute(td, 2*it+1);
   }
#ifndef CELL
   pthread_mutex_unlock(td->mutex);
#endif
}
```

Velocity Update 1

```
for(icache=0; icache<nxiter; icache++) {
   ixend = ixstart + pd->nxcache;
   if(ixend > ixterm) ixend = ixterm;
#ifdef CELL
   mem_prep_velocity(pd, ixstart, ixend);
#endif
   for(iz=nabc_top; iz<nzvec-nabc_bottom; iz++) {
#ifdef CELL
      mem_read_write_velocity(pd, ixstart, ixend, iz, nzvec);
      ixc = 2; gxc = 0;
#else
      ixc = ixstart+2; gxc = ixstart;
#endif
      for(ix=ixstart; ix<ixend; ix++) {
         v_vx[IZ][ixc] = compute_vx(v_vx, v_bx, v_txx, v_txz, v_g1, v_g2,
                         ixc, gxc, GZ, IZM1, IZ, IZP1, IZP2);
         v_vz[IZ][ixc] = compute_vz(v_vz, v_bz, v_tzz, v_txz, v_g1, v_g2,
                         ixc, gxc, GZ, IZM2, IZM1, IZ, IZP1);
         ixc++; gxc++;
      }
   }
#ifdef CELL
   mem_read_write_velocity(pd, ixstart, ixend, nzvec, nzvec);
   mem_read_write_velocity(pd, ixstart, ixend, nzvec+1, nzvec);
#endif
   ixstart += pd->nxcache;
   }
```

Pointer Macros

```
#ifdef CELL
#define IZM2 piv[0]
#define IZM1 piv[1]
#define IZ   piv[2]
#define IZP1 piv[3]
#define IZP2 piv[4]
#define GZ   pic[0]
#else
#define IZM2 iz-2
#define IZM1 iz-1
#define IZ   iz
#define IZP1 iz+1
#define IZP2 iz+2
#define GZ   gz
#endif
```

Velocity Update 2

```
inline VECFLOAT compute_vx(
   VECFLOAT **v_vx, VECFLOAT **v_bx,
   VECFLOAT **v_txx, VECFLOAT **v_txz,
   VECFLOAT v_g1, VECFLOAT v_g2,
   int ix, int gx, int gz,
   int izm1, int iz, int izp1, int izp2)
{
   return ADDV( v_vx[IZ][ix], MULV( v_bx[gz][gx],
          ADDV( MULV(v_g1, SUBV(v_txx[iz][ix+1],
                    v_txx[iz][ix-2])),
          ADDV( MULV(v_g2, SUBV(v_txx[iz][ix],
                    v_txx[iz][ix-1])),
          ADDV( MULV(v_g1, SUBV(v_txz[izp2][ix],
                    v_txz[izm1][ix])),
                MULV(v_g2, SUBV(v_txz[izp1][ix],
                          v_txz[iz][ix]))
                          )))));
}
```

Benchmarking Machines

Machine	Cores	Speed	Memory	Cache
PS3	1+6	3.2 GHz	.25 GB	.5 MB
QS20	2+16	3.2 GHz	1 GB	.5 MB
Intel	8	2.6 GHz	16 GB	6 MB

Benchmarks

Marmousi2 shot gather - 8 second simulation

Threads	PS3	QS20	Intel
1	31:39	19.24	30:29
2	16:10	10:25	17:17
3	11:08	7:58	17:27
4	9:14	7:21	14:57
5	9:06	8:49	16:21
6	7:40	6:30	14:27
7	N/A	5:01	14:08
8	N/A	6:41	13:37

Living With the Code 1

1. Memory access is regular and predictable.

Living With the Code 1

1. Memory access is regular and predictable.
2. Computational kernel is simple.

Living With the Code 1

1. Memory access is regular and predictable.
2. Computational kernel is simple.
3. Resulting SPE program is small.

Living With the Code 1

1. Memory access is regular and predictable.
2. Computational kernel is simple.
3. Resulting SPE program is small.
4. Methodology should generally be applicable to finite-difference codes.

Living With the Code 2

1. #defines are messy.

Living With the Code 2

1. #defines are messy.
2. Must maintain clear picture of abstractions while maintaining code.

Living With the Code 2

1. #defines are messy.
2. Must maintain clear picture of abstractions while maintaining code.
3. Consistent use of z pointer macros is essential.

Living With the Code 2

1. #defines are messy.
2. Must maintain clear picture of abstractions while maintaining code.
3. Consistent use of z pointer macros is essential.
4. Boundary conditions were rolled into main loop to facilitate a single pass.

Living With the Code 2

1. #defines are messy.
2. Must maintain clear picture of abstractions while maintaining code.
3. Consistent use of z pointer macros is essential.
4. Boundary conditions were rolled into main loop to facilitate a single pass.
5. Would be hesitant to expect someone else to maintain the code.

Conclusions

1. Portability between x86 and Cell is possible.

Conclusions

1. Portability between x86 and Cell is possible. (if not pretty)

Conclusions

1. Portability between x86 and Cell is possible. (if not pretty)
2. Programmer effort to manage the 'cache' results in significantly better scalability.

Conclusions

1. Portability between x86 and Cell is possible. (if not pretty)
2. Programmer effort to manage the 'cache' results in significantly better scalability.
3. Heterogeneous processors may become mainstream.

Conclusions

1. Portability between x86 and Cell is possible. (if not pretty)
2. Programmer effort to manage the 'cache' results in significantly better scalability.
3. Heterogeneous processors may become mainstream.
4. Will be compared to OpenMP implementation?

References

Levander, A. R., 1988, Fourth-order finite-difference P-Sv seismograms: Geophysics, **53**, 1425–1436.

Martin, G. S., R. Wiley, and K. J. Marfurt, 2006, Mamousi2: An elastic upgrade for marmousi: The Leading Edge, **25**, 156–166.