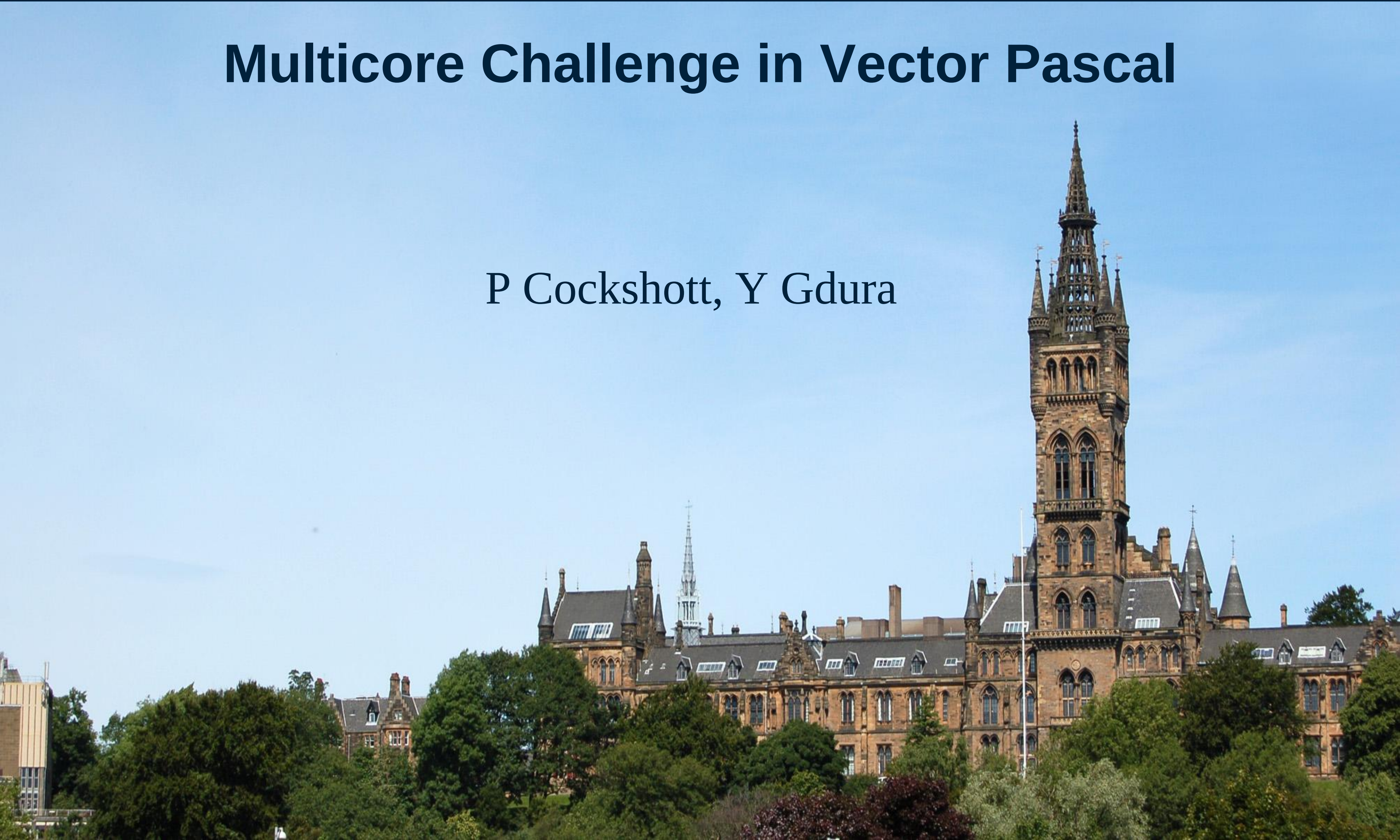
Multicore Challenge in Vector Pascal

P Cockshott, Y Gdura



N-body Problem

- Part 1 (Performance on Intel Nehalem)
  - Introduction (Vector Pascal, Machine specifications, N-body algorithm)
  - Data Structures (1D and 2D layouts)
  - Performance of single thread code (C and Vector Pascal)
  - Performance of multithread code (VP SIMD version)
  - Summary Performance on Nehalem

- Part 2 (Performance on IBM Cell)
  - Introduction
  - New Cell-Vector Pascal (CellVP) Compiler
  - Performance on Cell (C and Vector Pascal)

University of Glasgow

Vector Pascal

- **Extends Pascal’s support for array operations**
- **Designed to make use of SIMD instruction sets and multi-core**

Xeon Specifications

- Hardware
  - Year 2010
  - 2 Intel Xeon Nehalem (E5620) - 8 cores
  - 24 GB RAM, 12MB cache
  - 16 threads
  - 2.4 GHz
- Software
  - Linux
  - Vector Pascal compiler
  - GCC version 4.1.2

The N body Problem

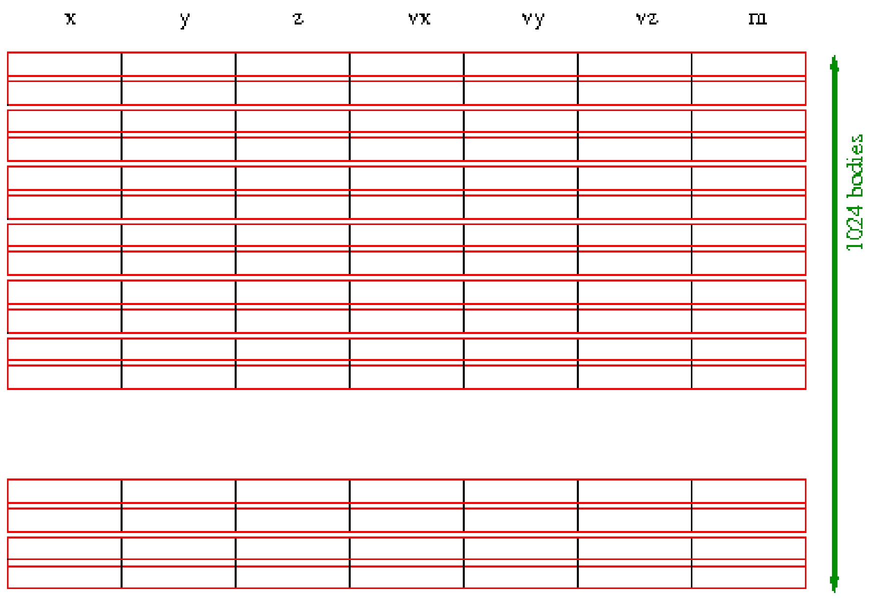
```
For 1024 bodies
Each time step
    For each body B in 1024
        Compute force on it from each other body
        From these derive partial acceleration
        Sum the partial accelerations
        Compute new velocity of B
    For each body B in 1024
        Compute new position
```

Data Structures

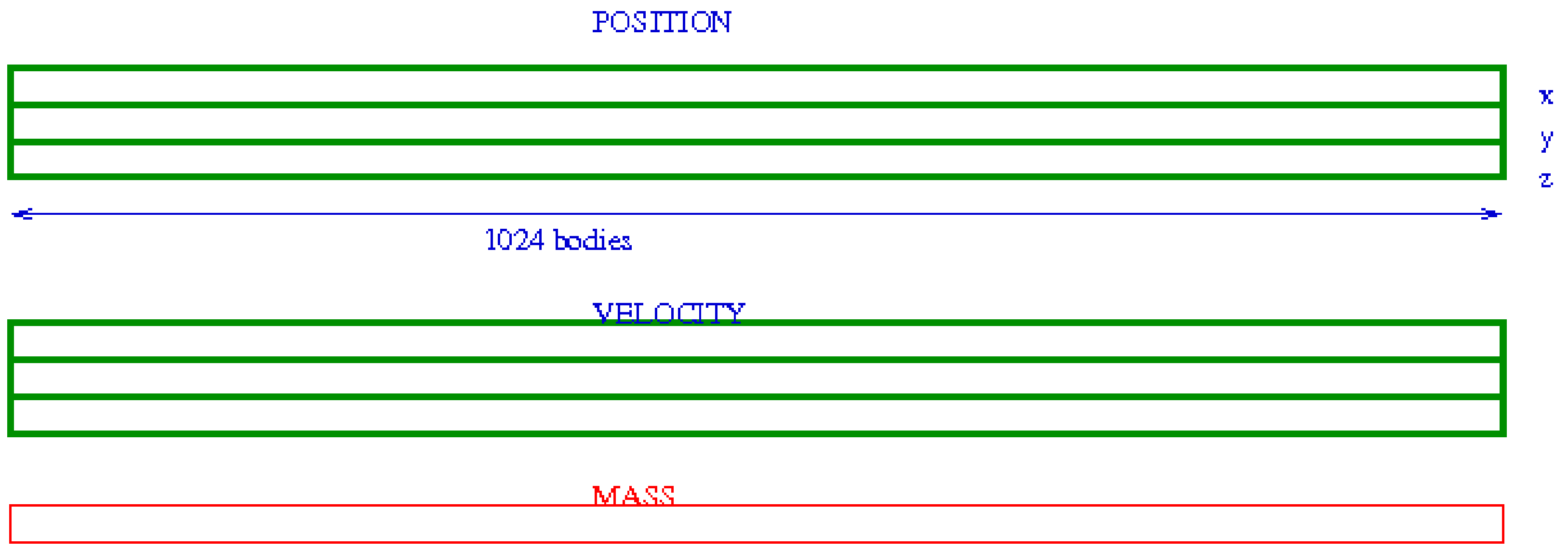
The C implementation stores the information as an array of structures each of which is

```
struct planet {
  double x, y, z;
  double vx, vy, vz;
  double mass;
};
```

Does not align well with cache or SIMD registers



Alternative Horizontal Structure



This layout aligns the vectors with the cache lines and with the vector registers

The Reference C Version

```
for (i = 0; i < nbodies; i++) {
    struct planet * b = &(bodies[i]);
    for (j = i + 1; j < nbodies; j++){
      struct planet * b2 = &(bodies[j]);
      double dx = b->x - b2->x;
      double dy = b->y - b2->y;
      double dz = b->z - b2->z;
      double distance = sqrt(dx * dx +
   dy * dy + dz * dz);
      double mag = dt / (distance *
   distance * distance);
      b->vx -= dx * b2->mass * mag;
      b->vy -= dy * b2->mass * mag;
      b->vz -= dz * b2->mass * mag;
      b2->vx += dx * b->mass * mag;
      b2->vy += dy * b->mass * mag;
      b2->vz += dz * b->mass * mag;
    }
```

Note that this version has side effects so the successive iterations of the outer loop can not run in parallel as the inner loop updates the velocities.

Equivalent Record Based Pascal

```
row:=0;
  b := planets[i];
  for j :=   1to n do begin
     b2 := planets[j];
     dx := b^.x - b2^.x;
     dy := b^.y - b2^.y;
     dz := b^.z - b2^.z;
     distance := sqrt(dx * dx + dy * dy + dz * dz);
     mag := dt*b2^.mass / (distance * distance * distance+epsilon);
     row[1] :=row[1]- dx     * mag;
     row[2] := row[2] -dy     * mag;
     row[3] :=row[3] - dz    * mag;
  end;
```

This is side effect free as the total change in the velocity of the ith planet is built up in a local row vector which is added to the planet velocities later.

Complexity and Performance Comparison

Timings below are for single threaded code on Xeon

	Vector Pascal	C
Unoptimised	28.9 ms	30 ms
-O3	23.5 ms	14 ms

Note: Pascal performs N^2 operations while C does $N^2/2$

SIMD friendly version – no explicit inner loop

```
pure function computevelocitychange(start:integer):coord;
```

-- declarations {M: pointer to mass vector, x: pointer to position matrix, di : displacement matrix, distance: vector of distances}

```
begin
    row:=x^[iota[0],i];
     { Compute the displacement vector between each planet and planet i.}
     di:=  row[iota[0]]- x^;
     { Next compute the euclidean distances }
     xp:=@ di[1,1];yp:=@di[2,1];zp:=@di[3,1]; { point at the rows }
     distance:= sqrt(xp^*xp^+ yp^*yp^+ zp^*zp^)+epsilon;
     mag:=dt/(distance *distance*distance );
     changes.pos:= \+ (M^*mag*di);
end
```

Row Summation operator builds x,y,z components of dv

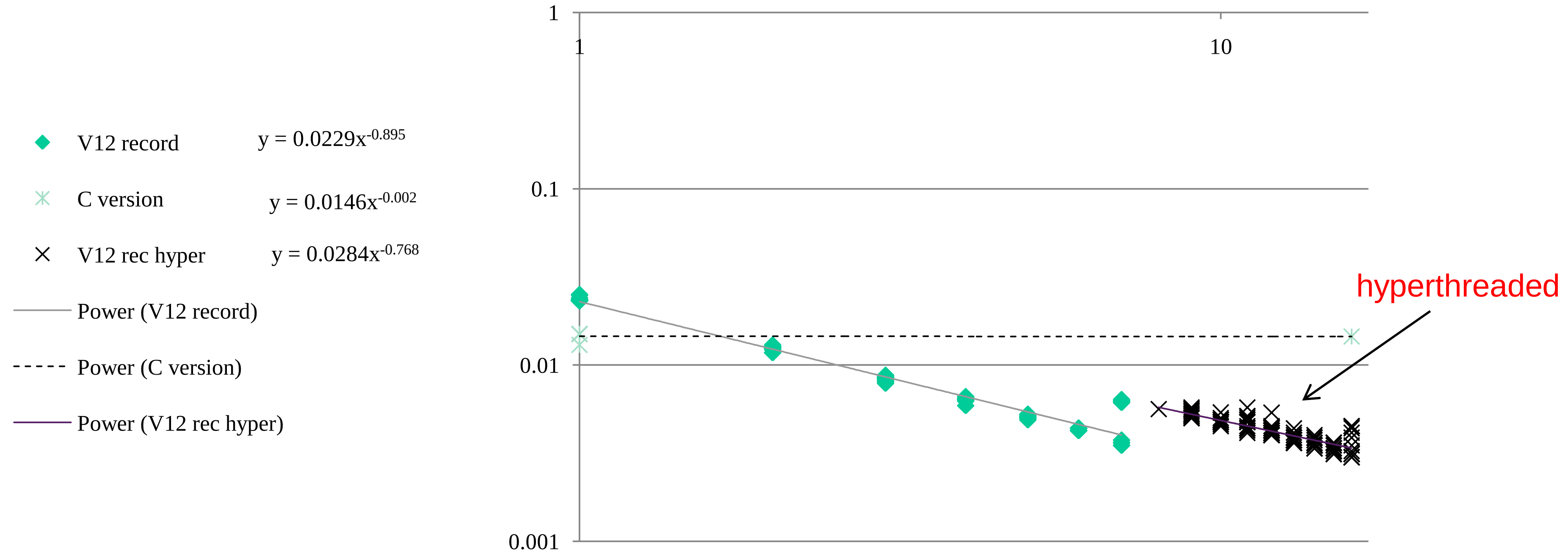
Pack this up in Pure Function Applied in Parallel

```
 procedure radvance(   dt:real);
 var dv:array[1..n,1..1] of coord; i,j:integer;
  pure function computevelocitychange(i:integer;dt:real):coord;
  begin
 {--- do the computation on last slide}
  computevelocitychange:=changes.pos;
  end;
begin
    dv :=computevelocitychange(iota[0],dt); { can be evaluated in
   parallel}
   for i:= 1 to N do                        { iterate on planets }
    for j:= 1 to 3 do                       { iterate on dimensions }
     v^[j,i]:=v^[j,i]+ dv[i,1].pos[j];      { update velocities }
   x^ := x^ + v^ *dt;                       { Finally update positions. }
end;
```

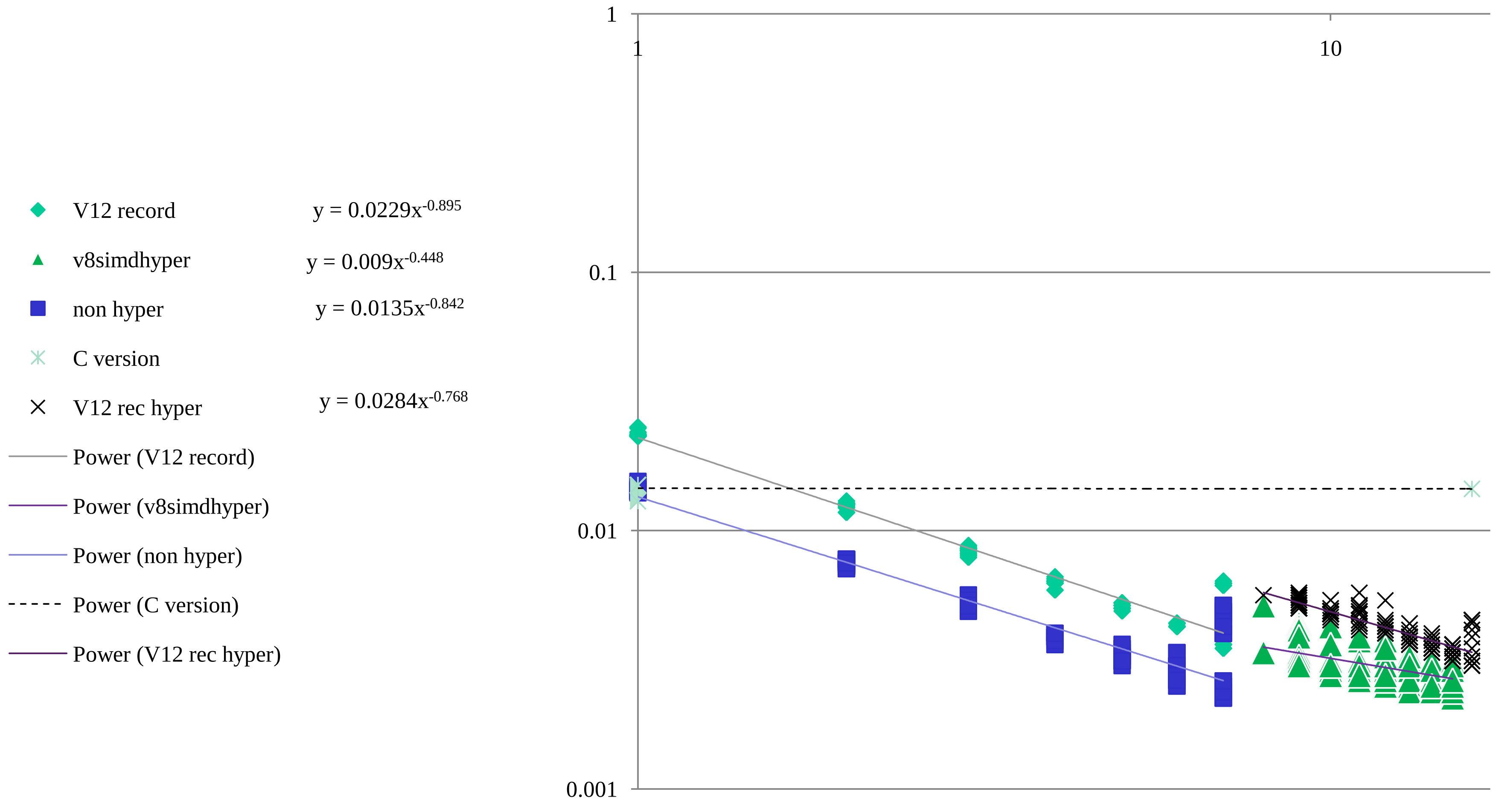
Iota[0] is the
0th index vector
If the left hand side

Now Compile with the Multiple Cores

- Programme unchanged compiled with from 1 to 16 cores for example
- vpc V12 –cpugnuP4 –cores8
- X axis threads, Y axis time in seconds, log log plot, 256 runs
- Mean time for 7 cores = 5.2 ms



Combined SIMD Multicore Performance



University of Glasgow

Summary Time per Iteration

Best performance on the Xeon was using 7 cores:

- SIMD performance scales as $c^{0.84}$
- Record performance scales as $c^{0.89}$,

  where c the number of cores.

	time
C optimised 1 core	14 ms
SIMD code Pascal 1 core	16 ms
SIMD code Pascal 7 cores	02.25 ms
Record code Pascal 1 core	23 ms
Record code Pascal 7 cores	03.75 ms

Performance in GFLOPS

- We pick the 6 core versions as it gives the peak flops, being just before the hyper-threading transition.
- This transition affects 7 and 8 thread versions.

Op.'s per Body	Vector Pascal	C
compute displacement	3	3
get distance	6	6
compute mag	5	3
evaluate dv	6	18
total per inner loop	20	30
times round inner loop	1024	512
times round outer loop	1024	1024
total per timestep	20971520	15728640

Language / version	Time	Number	GFLOPS	
	mec	Of Cores	Total	Per Core
		Xeon		
SIMD version Pascal	14.36	1	1.460	1.460
SIMD version Pascal	2.80	6	7.490	1.248
record version Pascal	23.50	1	0.892	0.892
record version Pascal	4.23	6	4.958	0.826
C version	**14.00**	**1**	**1.123**	**1.123**

Part 2

N-Body on Cell

- The Cell Architecture
- The CellVP Compiler using Virtual SIMD Machine
- Alignment and Synchronization
- Performance on Cell

The Cell Heterogeneous Architecture

- Year 2007
- Processors
  - **1 PowerPC (PPE) , 3.2 GHz, 512 MB RAM, 512KB L2, 32KB L1 cache**
  - **8 Synergistic processors (SPEs), 3.2 GHz, 256 KB**
- 2 Different Instruction sets (2 Different Compilers)
- Memory Flow Controller (MFC) on each SPE. (DMA, Mailbox, signals)
- Alignment boundary (16 bytes or 128bytes for better performance)
- Existing Supported Languages (C/C++ and Fortran)

The CellVP Compiler System

- *Objective*

  Automatic parallelizing compiler using virtual machine model

- *Aim at*

  Array expressions in intensive-data applications.

- *Built of*

  1. A PowerPC compiler
  2. A Virtual SIMD Machine (VSM) model to access the SPEs.

The PowerPC Compiler

- Transform sequential VP code into PPE code
- Convert large array expression into VM instructions
- Append to the prologue code, code to launch threads on the SPEs
- Append to the epilogue code, code to terminate SPEs’ threads.

Virtual SIMD Machine (VSM) Model

- VSM Instructions
  - Register to Register Instructions
  - Operate on virtual SIMD registers (1KB - 16KB)
  - Support basic Operations (+, - , / , * , sqrt , \+, rep ... etc)

VSM Interpreter

1. *The PPE Opcode dispatcher*

   i. Chops data equally on used SPEs
   ii. Formatting messages (opcode, registers to be used, starting address)
   iii. Writing messages to SPEs' Inbound mailbox
   iv. Waiting for a completion acknowledgment from SPEs (blocking mode)

2. *The SPE Interpreter* (A program runs in a background)

   i. Checks Inbound mailbox for new messages
   ii. On receiving a message, an SPE performs the required operation
   iii. Sends an acknowledgment with the completion , (If needed)

The CellVP Compiler System

1. Generates PowerPC machine instructions (sequential code)

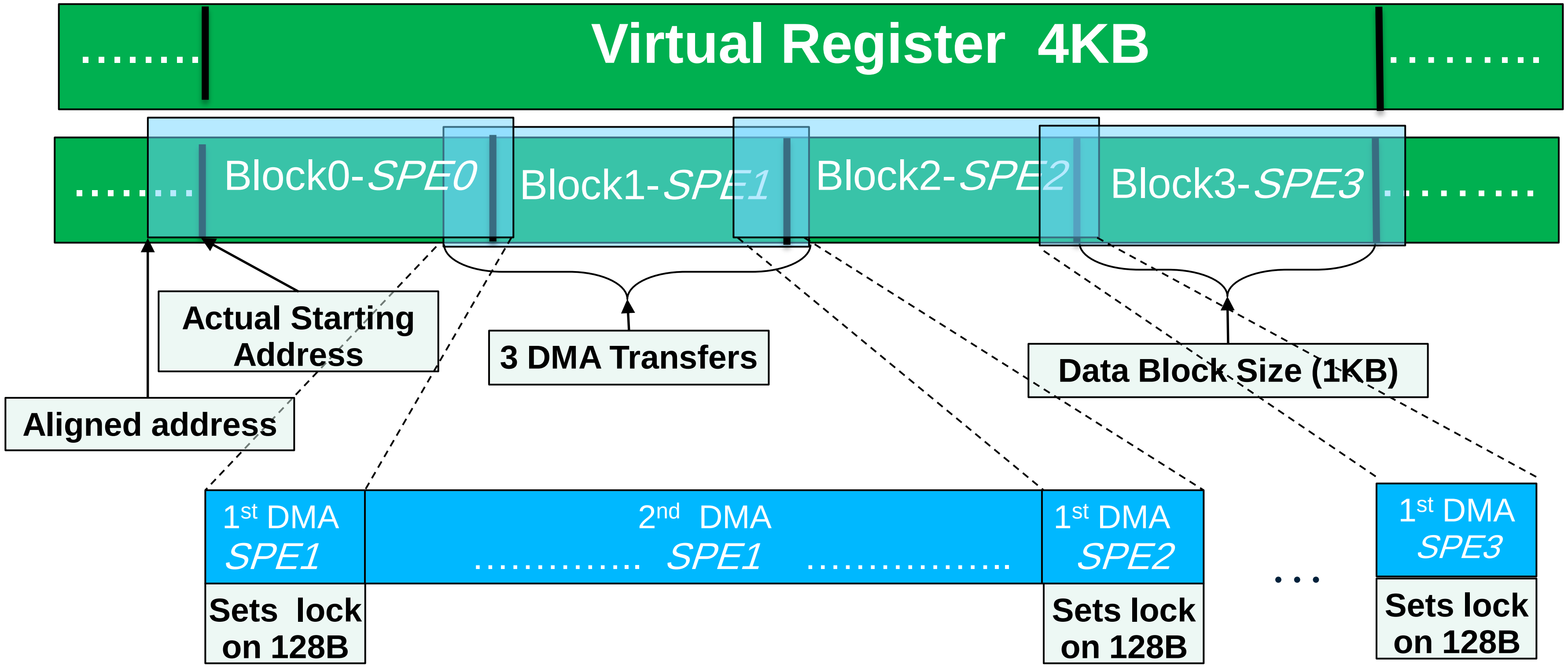
2. Generates VSM instructions to evaluate large arrays on SPEs.

3. PPE Handles
    1. Data Partitioning on SPEs
    2. Communication (Mailboxs)

4. SPE Handles
    1. Alignment   (load & Store)
    2. Synchronization

        Parts of data that might being processed on the preceding SPE and succeeding SPE

Alignment & Synchronization (Store Operation)



Virtual SIMD Register Chopped on 4 SPEs

N-Body Problem on the Cell

Code:	Same Xeon version
Data Structure:	large scale (4KB) Horizontal Structure
Machine:	PS3 (only four SPEs used)
Compilers:	GNU C/C++ compiler version 4.1.2
	Vector Pascal “CellVP”

Performance of VP&C on Xeon & Cell (GFLOPS)

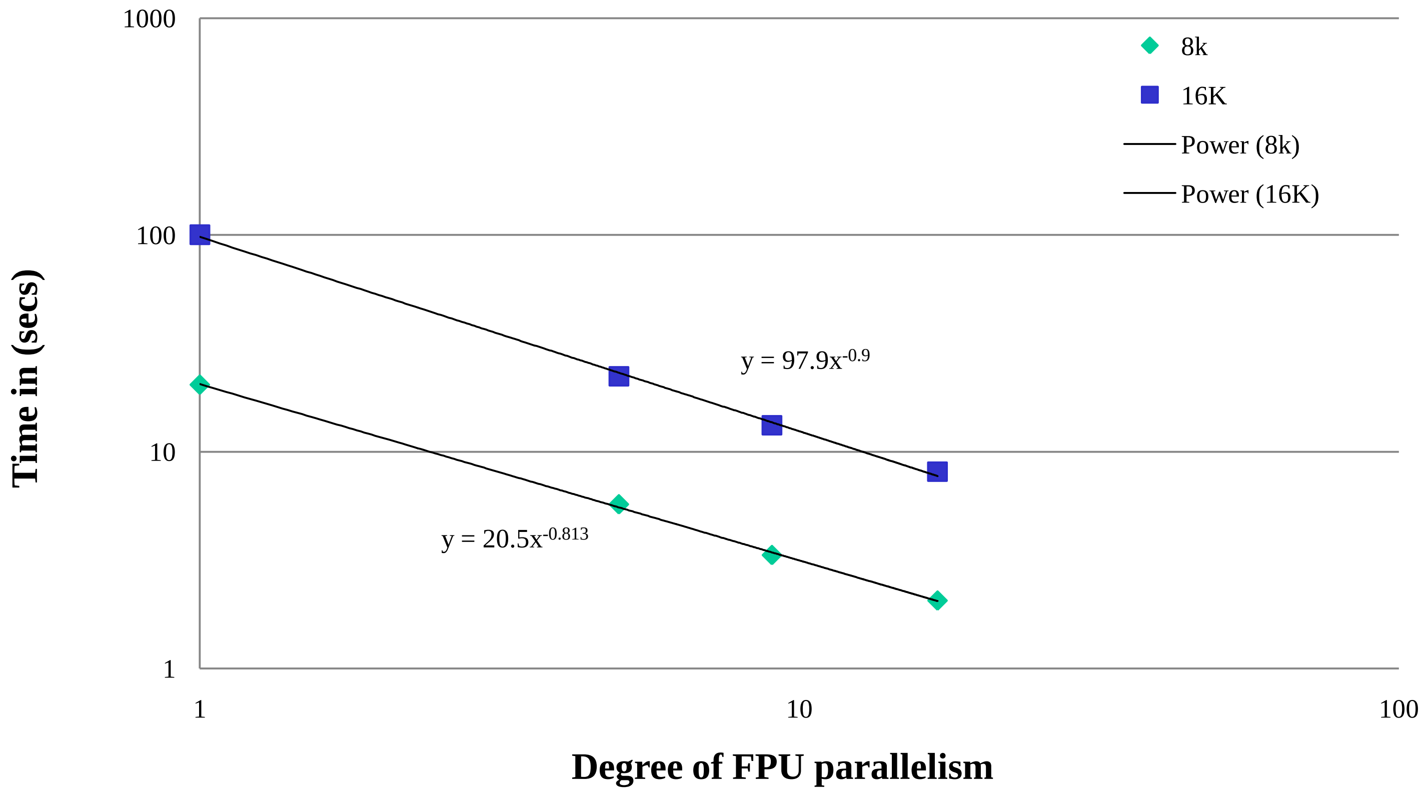
Op.'s per Body	Vector Pascal	C
compute displacement	3	3
get distance	6	6
compute mag	5	3
evaluate dv	6	18
total per inner loop	20	30
times round inner loop	1024	512
times round outer loop	1024	1024
total per time step	20971520	15728640

Language / version	Time msec	Number of Cores	GFLOPS Total	GFLOPS Per Core
Xeon				
SIMD version Pascal	14.36	1	1.460	1.460
SIMD version Pascal	2.80	6	7.490	1.248
record version Pascal	23.50	1	0.892	0.892
record version Pascal	4.23	6	4.958	0.826
C version	**14.00**	**1**	**1.123**	1.123
Cell				
Pascal (PPE)	381	1	0.055	0.055
Pascal (SPE)	105	1	0.119	0.119
Pascal (SPEs)	48	4	0.436	0.109
C (PPE, O3)	**45**	**1**	**0.349**	0.349

VP Performance on Large Problems

N-body Problem Size	Performance (seconds) per Iteration				
	Vector Pascal				C
	PPE	1 SPE	2 SPEs	4 SPEs	PPE
1K	0.381	0.105	0.065	0.048	0.045
4K	4.852	1.387	0.782	0.470	0.771
8K	20.355	5.715	3.334	2.056	3.232
16K	100.250	22.278	13.248	8.086	16.524

Log log chart of performance of the Cell



THANK YOU

ANY?