

Dyanamic Profiling and Load Balancing

**of an N-body Computational Kernel on
Heterogeneous Architectures**

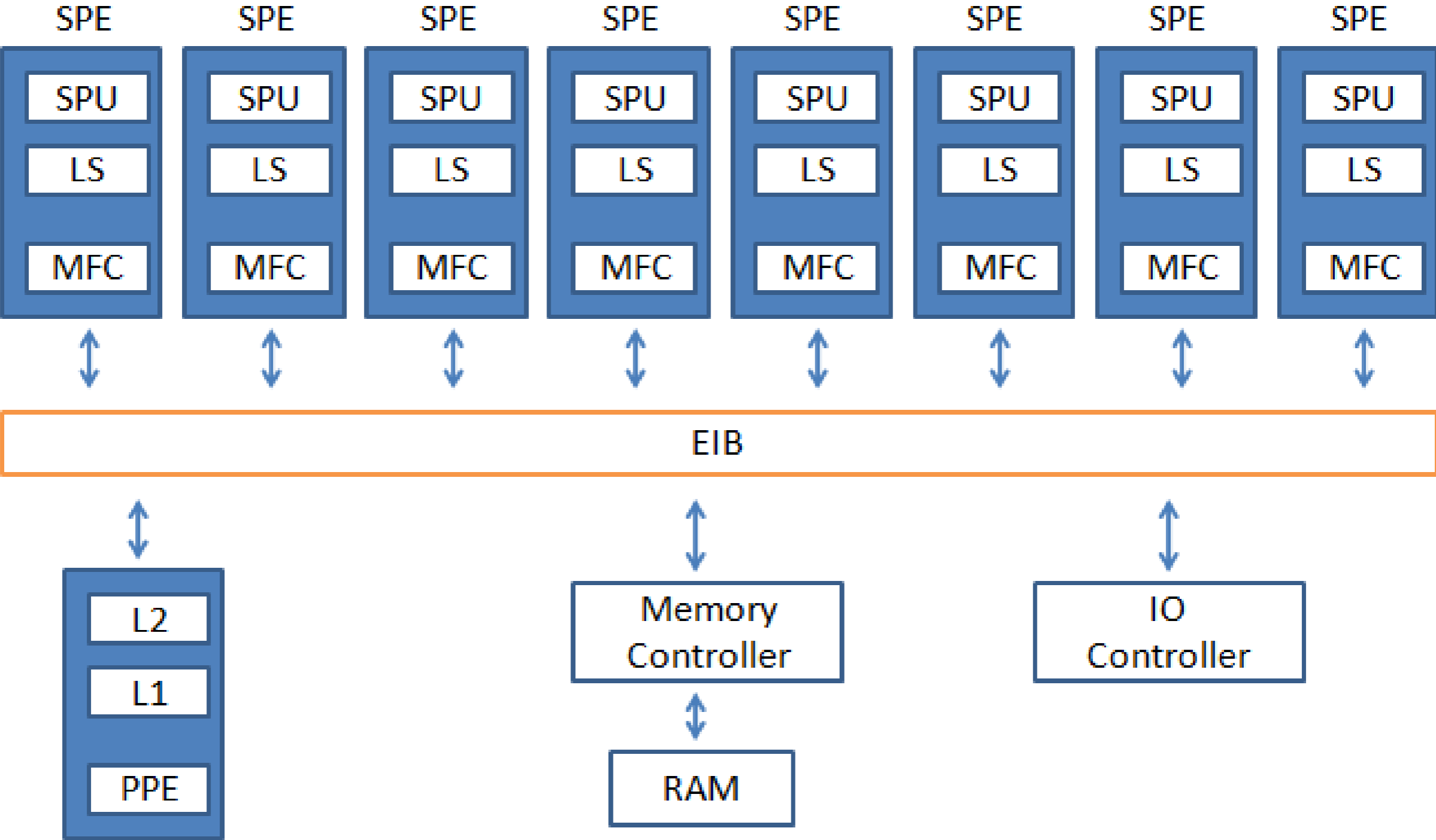
**GPUs and Accelerators in HPC Daresbury 28/29
Sept**

**Andrew Attwood and John Brooke
The University of Manchester**

Project overview

- **MSc project in advanced Computer Science**
- **Aim to investigate porting of numerically computational kernel to multicore architectures**
- **Cell Processor and GPUs chosen as complementary architectures**
- **Focus on computer science view, development of tools and methods**
- **Student chose to tackle N-body computational kernel.**

Cell Processor architecture



Workflow on Cell Processor

Full OS only on the PPE which organises computational work tasks on the SPEs

`spe_image_open()`
Load the image from the disk into main memory
return pointer to the image.

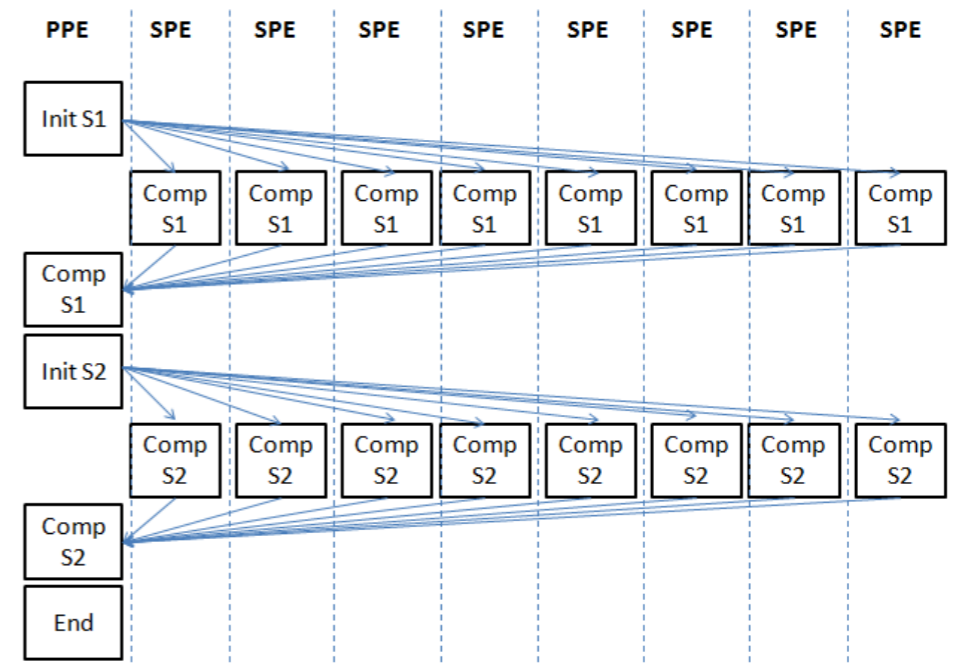
`spe_context_create()`
Create the context populating the supplied
`spe_context_ptr_t` structure

`spe_program_load()`
Load the supplied image pointer to the supplied
SPE defined by its context.

`spe_context_run()`
Run the current image on the context provided

`spe_context_destroy()`
Destroy the SPE context

`spe_image_close()`
Remove the image from main memory



Nehalem + Nvidia GPU

Similarly GPUs are designed for floating point calculations with simple logic.

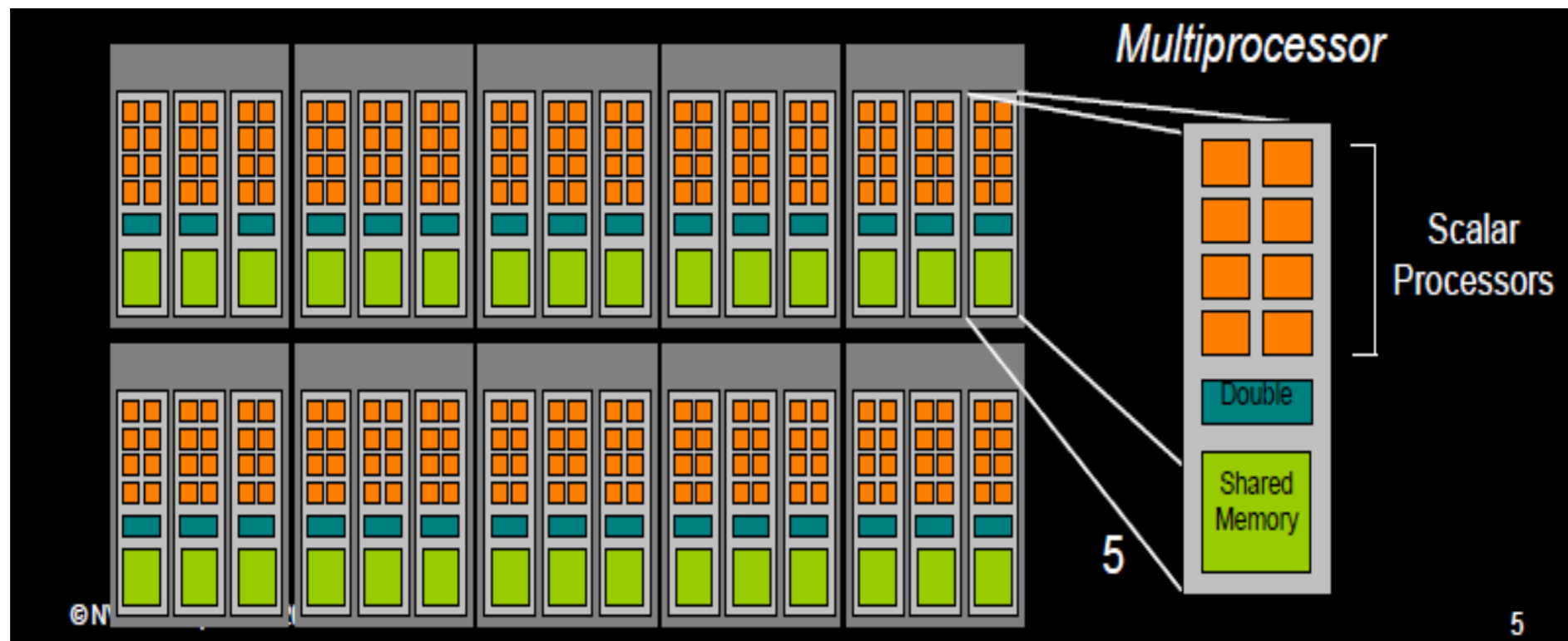
Combining them with a multi-threaded CPU allows task distribution so that the SIMD model can be used.

Data transfer to the GPU processing units has improved rapidly.



Massively threaded model for GPU processing

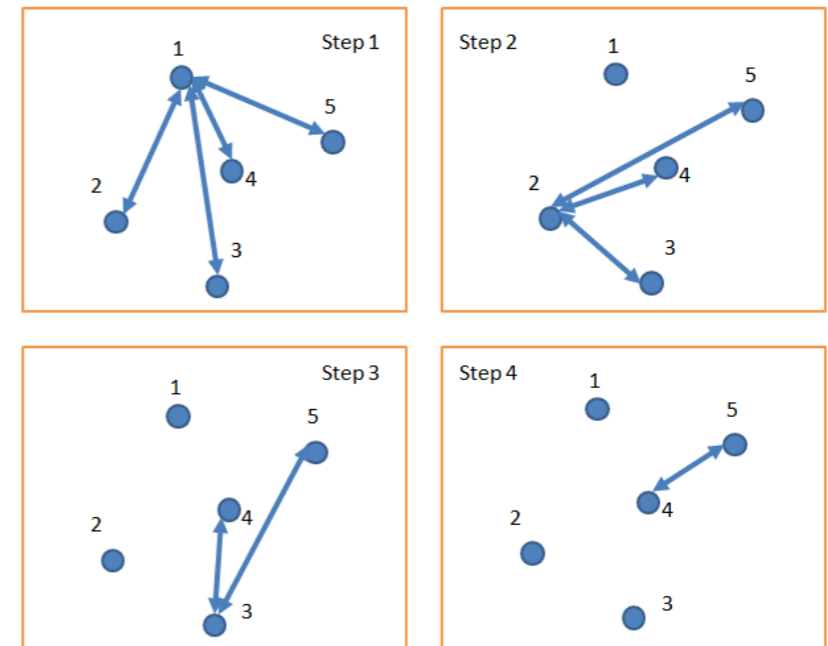
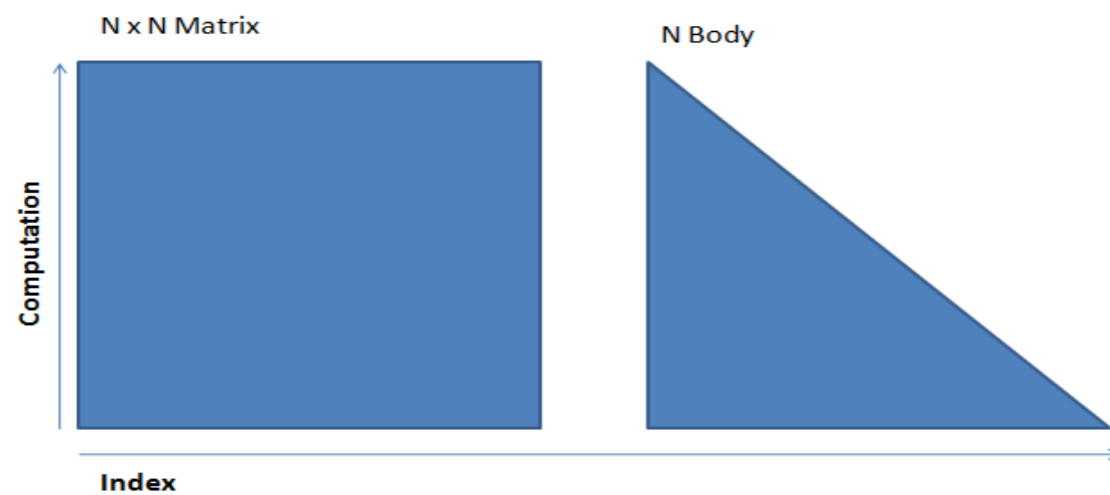
Processing uses a massively threaded model. Thread processing units grouped in units that have shared memory. Threads are grouped into blocks of 32 threads (a warp) that perform the same instruction on multiple data values.



Complexity of the N-body kernel

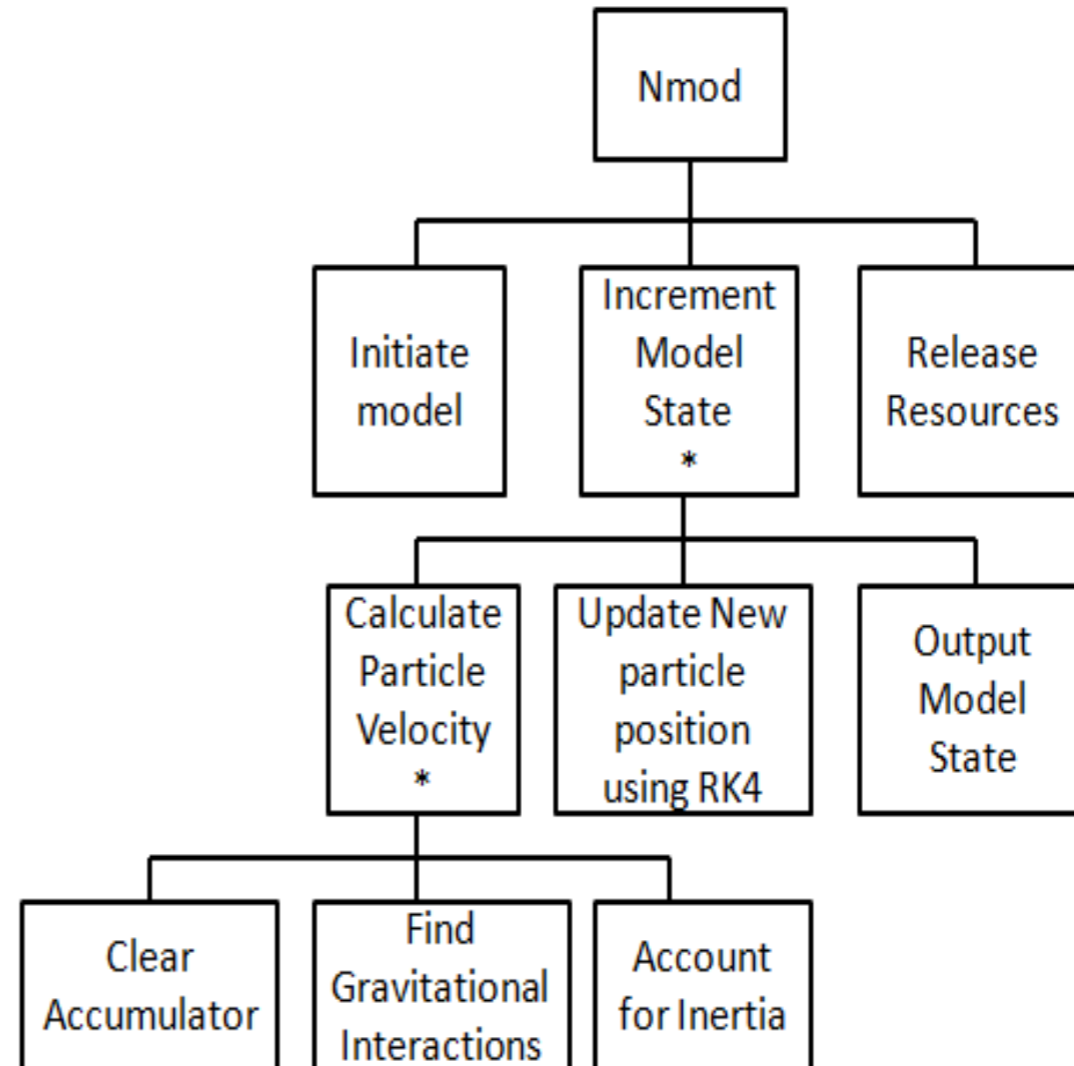
The N-body calculation has complexity of $N*(N-1)/2$ i.e. $O(N^2)$

This will need to be repeated at each step as the positions of the particles evolve.

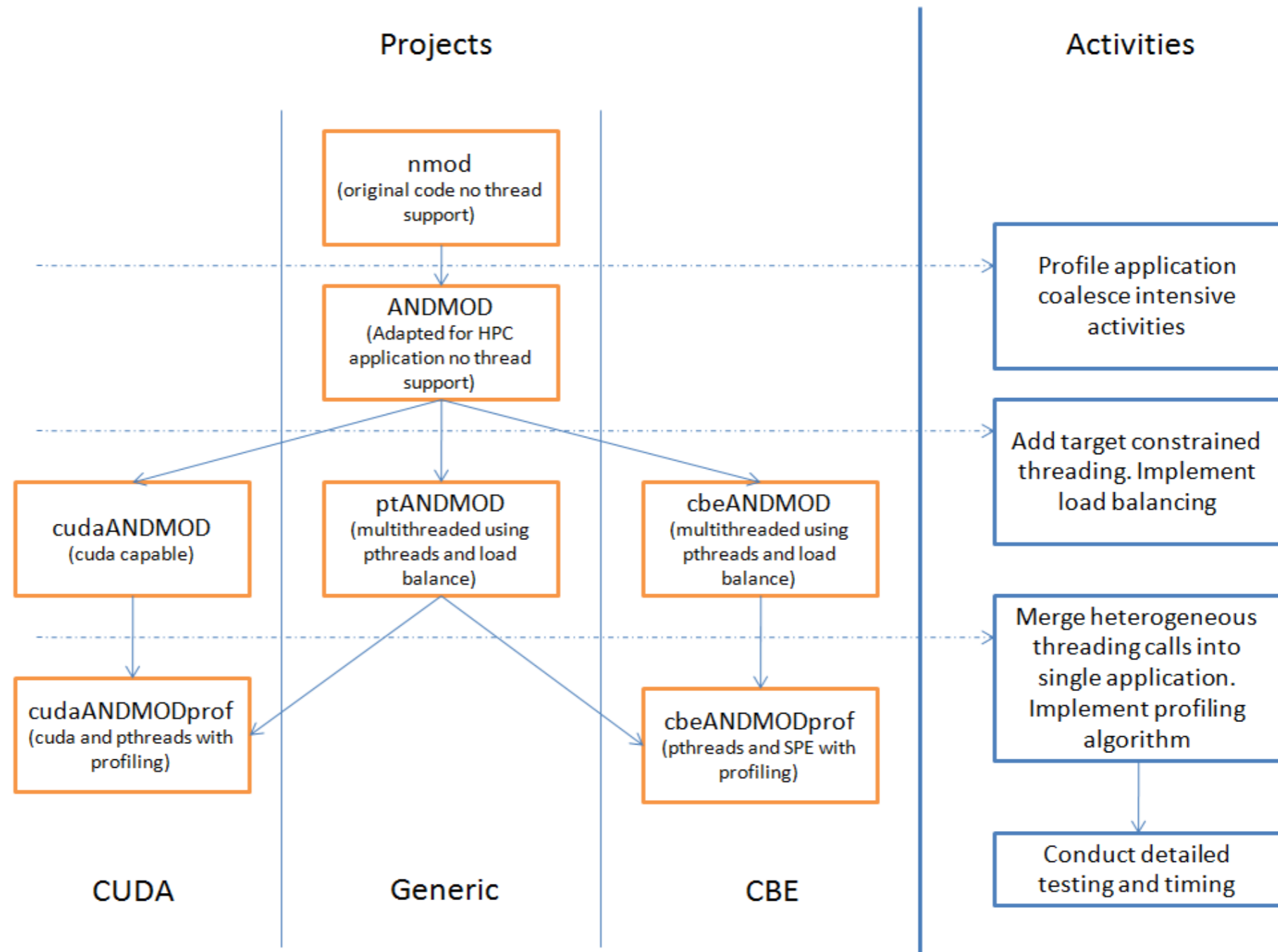


Task Diagram for the N-body kernel

□ State of the gravitational Interactions needs to be updated as solution evolves dynamically



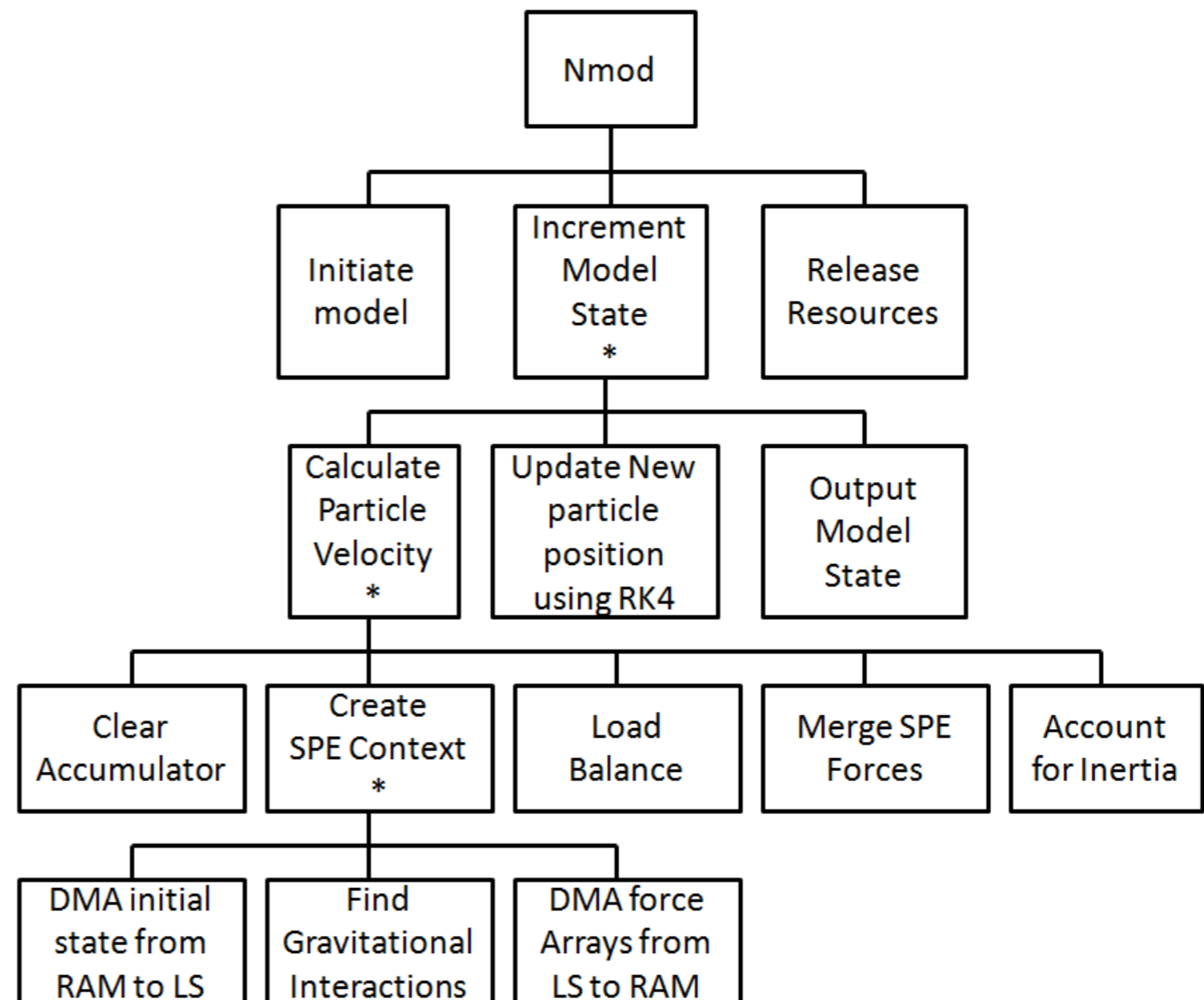
Project Plan



Adaptation of kernel for Cell Processor

Profiling indicated that the gravitational interactions took over 85% of computational time with 60 particles and this increased with numbers of particles.

This indicates that this part of the code needs to be calculated in parallel on the six SPEs

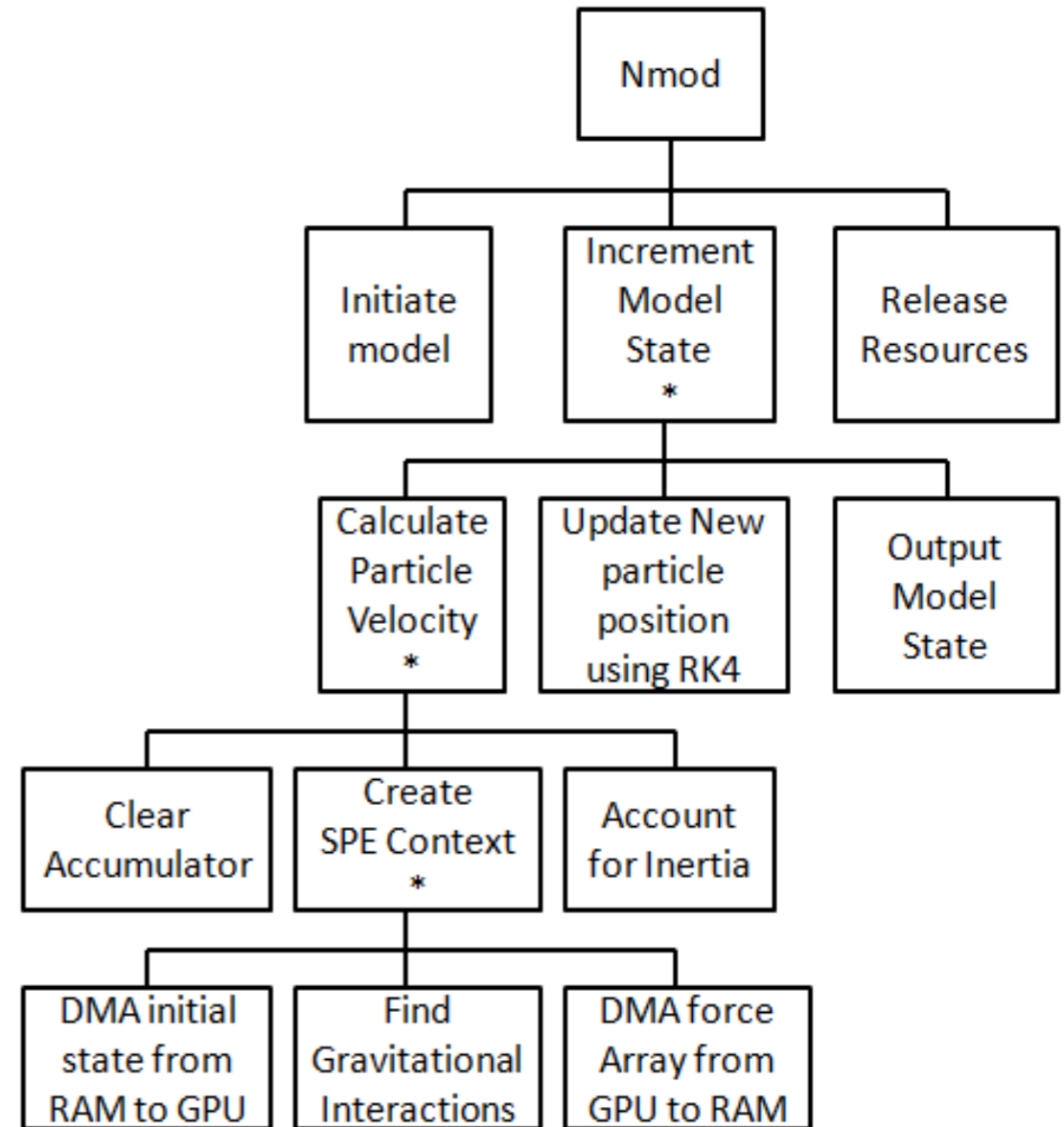


Adaptation to a GPU

The difference with the GPU is that each multi processing unit can access the same model.

On the version of the GPU used it was not possible to lock the values for a double for long range force accumulation.

We give each thread unit only local values, this results in doubling the number of calculations. On a massively threaded model this is acceptable.



Platforms

□ **CELL development was conducted on a 60GB Version 1 PS3 which had the other OS facility enabled. Sony commissioned Fixstars to develop a Linux variant which have the ability to operate on the PS3 and to accordingly give it access to the individual SPE. The PS3 system was installed with this Linux variant Yellow Dog Version 6.1.**

Initially, the application was developed on a dual core Xeon processor with an Nvidia Geforce 9500GT; however, this lacked the double precision required by the Nmod application. Due to this failing, the final implementation and development had to be conducted on the Mcore Server provided by the School of Computer Science at Manchester University.

SMP Code for gravitational interaction

```
Void * findgravitationalinteractions(void * sendin)
{
    trans * send = (trans * )sendin;
    double G = send->G;
    double * xloc = send->xloc;
    double * yloc = send->yloc;
    double * zloc = send->zloc;
    int speid = send->speid;
    double * xforce = send->xforce;
    double * yforce = send->yforce;
    double * zforce = send->zforce;
    double * mass = send->mass;

    int start = speid * ((num_particles - ( num_particles
%NUM_OF_THREADS)) / NUM_OF_THREADS);

    int end = start + ((num_particles - (num_particles %
NUM_OF_THREADS))/NUM_OF_THREADS);

    int remain = num_particles % NUM_OF_THREADS;

    int count = 0;
    if(remain > 0)
    {
        if(speid < remain)
        {
            start = start + speid;
            end = end + (speid + 1);
        }
        else
```

```

    {
        start = start + remain;
        end = end + remain;
    }
}

start = start + send->soff;
end = end + send->eoff;

for(int j = start;j < end;j++)
    {
        for(int i = j + 1;i < num_particles;i++)
            {
                .....
                //Calculate forces
                .....
            }
        }
send->work = count;
}
```

Load balancing and profiling

We use a load balancing approach based on a block cyclic approach to scheduling

Offsets are used in the SMP code to identify the work portions

In the testing part various combinations of cores (Cell) or threads (GPU) are tested for a single step and the minimum time result is stored

This can be used for the calculation that finds the relaxation towards equilibrium (lowest energy state)

Selecting the optimum plan

```
struct testplan_el{
    int gpu;
    int corenum;
    int testcount;
    struct testplan_el * up;
};
typedef struct testplan_el testplan;

void addcpu(cpu * cpulist,int proc,unsigned long long time,int
type,loadbal * temploadbal)
{
    while(cpulist->up != NULL)
    {
        cpulist = cpulist->up;
    }
    cpulist->up = (cpu *)malloc(sizeof(cpu));
    cpulist->proc = proc;
    cpulist->time = time;
    cpulist->type = type;
    if(temploadbal != NULL)
    {
        cpulist->loadbalist = temploadbal;
    }
    else
    {
        cpulist->loadbalist = NULL;
    }
    cpulist = cpulist->up;
}
```

```
for(int i = 1;i <= MAX_NUM_OF_THREADS;i++)
{
    testnum++;
    lasttest = addtest(testplanlist,0,i, num_particles/8);
}
testnum++;
lasttest = addtest(testplanlist,1,0,2);
```

Manual Results for Cell

Results for PPE on only

Body Count	No Thread Single PPE	P-threads version 2 PPE Cores
10	0m3.919s	0m5.745s
20	0m12.543s	0m11.304s
30	0m25.744s	0m20.399s
40	0m43.516s	0m32.868s
50	1m6.014s	0m48.016s
60	1m32.817s	1m7.470s
70	2m4.621s	1m28.534s
80	2m40.618s	1m52.760s
90	3m21.923s	2m20.231s
100	4m6.584s	2m52.290s
150	9m1.728s	6m28.314s
200	15m51.960s	10m57.586s
300	35m24.489s	23m42.368s

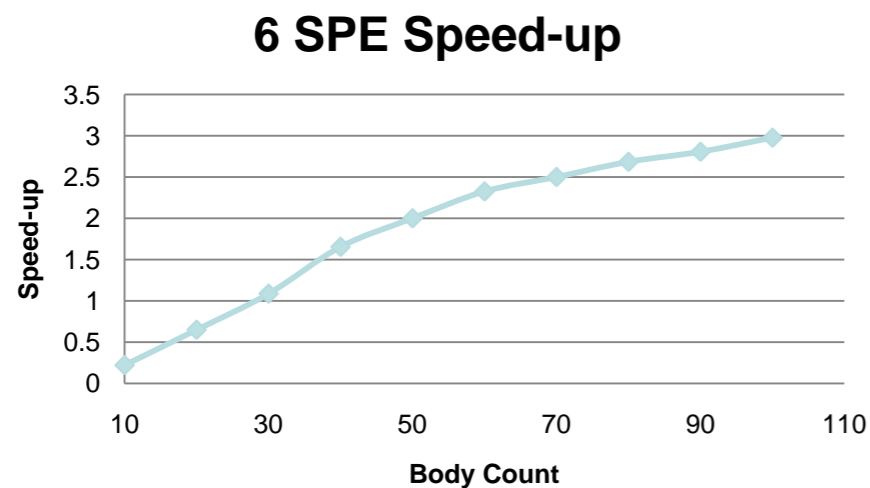
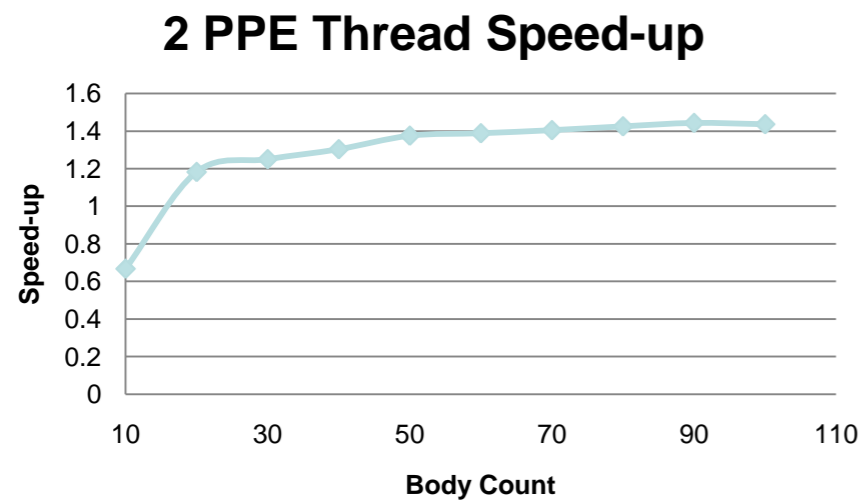
Results for SPE

Body Count	SPE VERSION	SPE VERSION	SPE VERSION	SPE VERSION	SPE VERSION	SPE VERSION
	1 SPE Enabled	2 SPE Enabled	3 SPE Enabled	4 SPE Enabled	5 SPE Enabled	6 SPE Enabled
10	0m7.375s	0m7.195s	0m10.943s	0m12.175s	0m15.100s	0m18.150s
20	0m17.680s	0m17.240s	0m17.691s	0m16.895s	0m17.770s	0m20.497s
30	0m33.538s	0m31.097s	0m27.985s	0m25.021s	0m21.633s	0m23.180s
40	0m55.171s	0m49.290s	0m43.228s	0m37.048s	0m30.615s	0m26.457s
50	1m22.507s	1m13.314s	1m1.836s	0m49.964s	0m38.446s	0m32.706s
60	1m55.380s	1m36.857s	1m21.442s	1m5.053s	0m47.707s	0m39.958s
70	2m34.893s	2m9.264s	1m47.145s	1m22.028s	0m57.631s	0m50.261s
80	3m18.785s	2m45.077s	2m12.924s	1m40.813s	1m8.020s	1m0.293s
90	4m10.373s	3m26.871s	2m45.861s	2m6.758s	1m26.089s	1m12.030s
100	5m5.908s	4m11.521s	3m21.242s	2m29.335s	1m37.942s	1m23.275s

Autoprofiling compared with manual results

Autoprofiling does not detect the PPE option for the lowest particle counts. However it detects the crossover points for SPEs

Auto Profiling Algorithm	
Body Count	Best Number of SPE
10	2 SPE
20	4 SPE
30	5 SPE
40	6 SPE
50	6 SPE
60	6 SPE
70	6 SPE
80	6 SPE
90	6 SPE
100	6 SPE
150	6 SPE
200	6 SPE



Auto-profiling of Nehalem/GPU

Autoprofile
of Nehalem with
and without the
GPU cores.

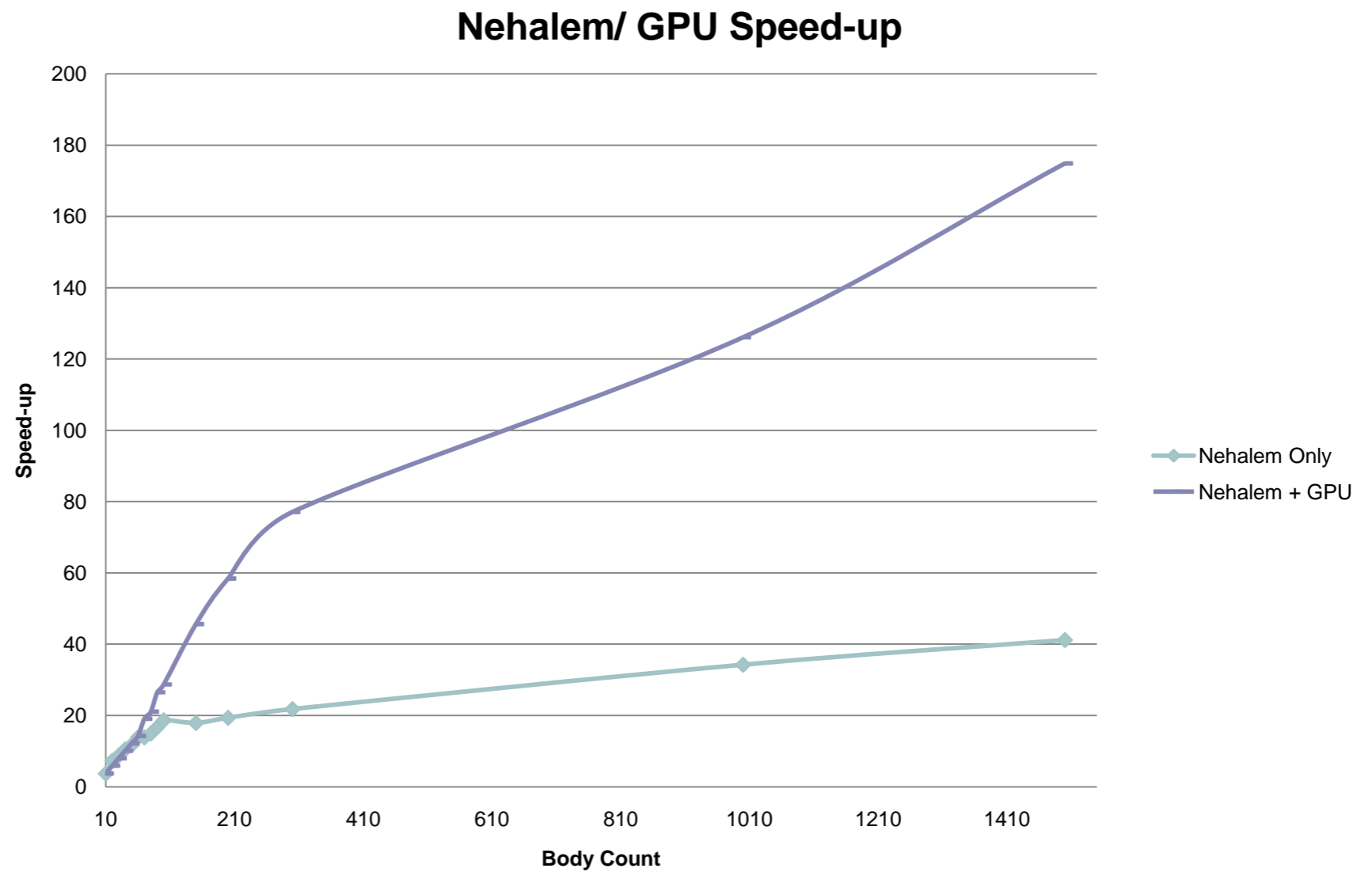
Nehalem only
speedup
superlinear owing
to cache effects

GPU solution 190X
speedup on 1500
bodies

N-body Count	Single Thread	8 Nehalem Cores Auto Profiling Algorithm		Nvidia GPU and 8 Nehalem Cores Auto Profiling Algorithm	
	Time	Time	Best Core	Time	Best Processor/core
			Target		Target
10	0m3.743s	0.1.011s	2	0.1.022s	2
20	0m11.809s	0.1.629s	2	0.1.637s	2
30	0m23.971s	0.2.743s	2	0.2.672s	2
40	0m40.320s	0.3.853s	2	0.3.790s	2
50	1m0.939s	0.5.210s	2	0.5.103s	2
60	1m25.460s	0.6.167s	2	0.5.857s	GPU
70	1m54.531s	0.8.270s	3	0.6.610s	GPU
80	2m27.401s	0.9.868s	3	0.7.231s	GPU
90	3m5.499s	0.11.094s	3	0.7.683s	GPU
100	3m49.638s	0.12.312s	3	0.8.186s	GPU
150	8m21.844s	0.28.057s	5	0.10.858s	GPU
200	14m36.327s	0.45.314s	6	0.15.126s	GPU
300	32m7.590s	1.28.506s	7	0.25.053s	GPU
1000	390m52.289s	11.25.329s	7	3.6.300s	GPU
1500	868m34.088s	21.6.735s	8	4.57.640s	GPU

Graphical View of autoprofiling

Crossover point
for using GPU at
60 bodies



Future work

- **We could store previous optimum states to narrow down the extent of the search for the optimum configuration.**
- **OS enhancements – information about where the threads are physically running – important in clouds.**
- **Take account of speed of approach to optimum configuration, extra tests provide more information far from optimum.**
- **Use of machine learning, our algorithm assumes a very simple relation between underlying algorithm and its physical implementation (offset value via block cyclic scheduling strategy).**