Technical Report CSTN-077

Benchmarking GPU Devices with N-Body Simulations

D.P. Playne, M.G.B. Johnson and K.A. Hawick
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
d.p.playne@massey.ac.nz, mitchelljohnson@ihug.co.nz, k.a.hawick@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

Abstract

Recent developments in processing devices such as graphical processing units and multi-core systems offer opportunities to make use of parallel techniques at the chip level to obtain high performance. We discuss the difficulties in establishing suitable benchmark codes for making comparisons across these device architectures and in a way that is representative of key applications. We report on our use of classical dynamical particle collision simulation codes as benchmarks for comparing modern GPUs. We discuss our findings in terms of architectural features for parallelism as well as clock speed issues.

Keywords: benchmarking and measurements; multi-thread; multi-core; performance analysis.

1 Introduction

Benchmarking architectures at the chip level in a manner to give indications of high level application performance is a non-trivial problem. Low level benchmarks concerning raw memory access times or speeds of individual floating point operations are easy to quantify but can give misleading estimates on the overall performance of an algorithm as it might appear in a high level application such as a modern game, animation or simulation.

Devices such as Graphical Processing Units (GPUs) and multi-core CPUs are becoming very widely available and are often an important component for delivering high performance on a gaming or simulation platform. We have used a number of GPUs for accelerating various scientific simulations [1] that are representative of some of the algorithms and ideas in physics and gaming engines [2, 3]. With the recent development of General-Purpose computation on GPU (GPGPU) technologies [5–10], a large number of scientific simulations are being implemented to utilise the cheap computational power of GPUs.

While many benchmarks exist for GPUs, they often focus on either graphical rendering or the individual capabilities of the graphics card. A GPGPU benchmark that evaluates a GPUs performance for scientific simulations is required. The N-body particle model was chosen as a benchmark as it represents the requirements of many simulations. This is an N^2 problem that requires memory access patterns that are representative of many simulations.

In a scientific simulation, these interactions might involve hard-body mechanisms or soft-force models based on an empirically fitted potential energy model such as Lennard-Jones [4]. Particle models find use in games for modelling sand and other physically based particles but also for approximating water flow or even for providing a substrate model for moving cloth and other materials. Other uses of rigid body models include providing an approximating skeleton for the limbs of animated figures and other objects in a game.

In this paper we describe some benchmarking issues in Section: 2. We review issues associated with classical dynamical particle algorithms in Section: 3, their implementation in Section: 4 and report on some specific benchmarking data obtained using these ideas on various CPUs and GPUs in Section: 5. In Section: 6 we discuss how a fair comparison can be made across these quite different devices, and explore the implications for some of the architecture features involved. We offer some ideas for further device level benchmarks and concluding remarks in Section: 7.

2 Benchmarking Processing Devices

Relatively recent developments in new processing devices include the move to 64-bit architectures, the incorporation of multiple cores onto CPUs and the introduction of specialist devices such as GPUs for specific processing tasks. The move to 64-bit architectures has had implications for games and simulations [14] in terms of floating point performance but has perhaps had more impact on applications complexity in terms of data storage and addressing. For the most part, 64-bit issues can be taken care of by the compiler and by appropriate changes in operating system level code.

Multi-core issues however continue to challenge applications programmers [15] and typically require the use of specific parallel programming models such as multi-threading or of application languages with in-built parallel constructs such as High Performance Fortran [16]. Assessing or benchmarking the performance of applications on parallel systems in a fair manner across widely differing systems has proved a notoriously hard problem to solve. The NAS parallel Benchmarks [17] provided a partial solution to this problem by abstracting low level details away and specifying a relatively high level algorithm and problem specification rather than just providing a specific low level code implementation that vendors and other benchmarkers would have to optimise.

3 N-Body Particle Algorithms

Particle simulations involve computing the motion of a number of particles (defined by a position, velocity, mass and possibly a shape). These particles move according to Newton's Laws of Motion [23] and often attract/repluse each other according to a potential function. As is common for particle simulations, the potential function used for this simulation is Newton's Law of Universal Gravitation [23]. As defined by:

$$F = \frac{Gm_1m_2}{r^2} \quad (1)$$

The total force on each particle can be calculated from the total sum of each of the forces the other particles impart on it. A particle simulation must calculate the total force on each particle and then apply a suitable numerical integration method to calculate the change in velocity and position of each particle over a discrete time-step. These particle simulations can be utilised for many purposes, but as a benchmark simply performing the calculations is sufficient. However, there are several algorithms available for calculating the total force on each particle. Figure: 1 shows a screen shot of an example particle simulation.

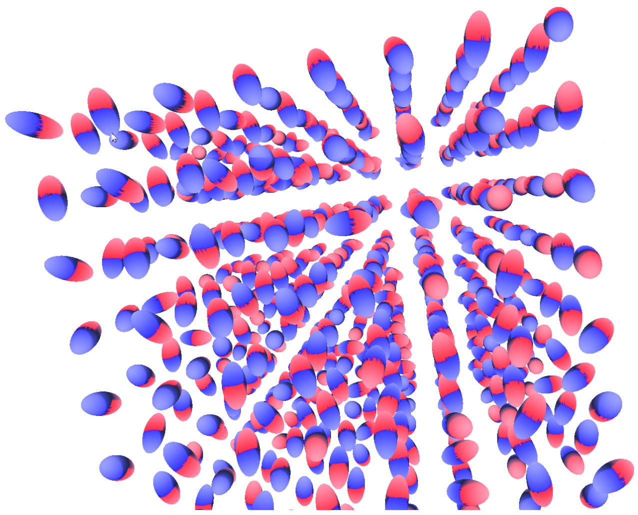

Figure 1: A screenshot of an example particle simulation.

All-Pairs Method The All-Pairs method is the simplest algorithm for calculating the total force on each particle. The force on every particle (as defined by the potential function of the simulation) from each other particle is calculated one interaction at a time and the final forces are used to calculate the change in velocity and position of the particle. This is an $O(N^2)$ algorithm as for N particles, the total acceleration on each particle requires N calculations. This method is simple to implement but is limited by the exponential computational complexity of the algorithm. There are approximation methods for reducing this complexity from $O(N^2)$ to $O(N\ log\ N)$.

Barnes-Hut Treecode One common method for performing N-body force calculations in $O(N\ log\ N)$ time is the Barnes-Hut tree method [24–26]. This method is based on the work by Andrew Appel [27] and is more commonly used due to its greater performance [25]. The method approximates the interactions of the system to allow them to be calculated faster at the cost of a controllable loss of precision.

The interaction between a single particle and a cluster of distant particles can be approximated by a single interaction calculation. The combined force a cluster exerts on the single particle can be calculated by averaging the positions and summing the weights of the particles and then performing a single potential calculation. This method works well and the approximation is accurate enough to work when the distance between the particle and the cluster is large.

Barnes and Hut use Octrees to define clusters and calculate an approximation to a controllable degree of error. Octrees are built by subdividing a three dimensional cube into eight smaller cubes, this structure is defined as a tree (each node has eight children thus the name octree). When building a octree for a particle simulation, each node in the tree (a cube of space) is divided into eight smaller nodes until each node is either empty or contains one particle. See Figure: 2 for an example Octree.

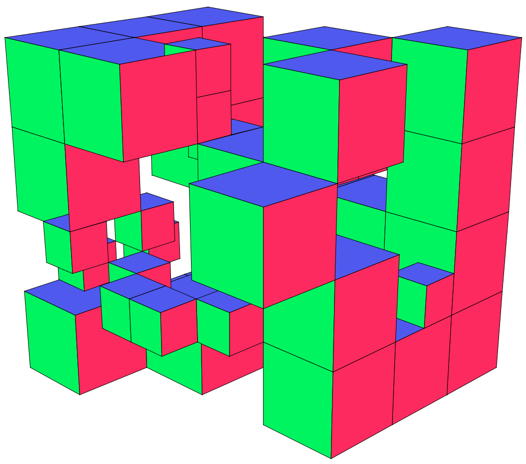
Figure 2: A visualisation of an Octree, only nodes containing a particle are rendered.

The method Barnes and Hut suggest for constructing this tree is via an insertion method which allows the construction of a tree with a time complexity of $O(N\ log\ N)$ [24]. Once the space of the simulation has been subdivided, each node is recursively assigned a mass and centre-of-mass according to the either real particle it contains or the particles within its sub-nodes.

The algorithm moves recursively down the tree and if the length of the current node l over the distance D to the current particle is less than θ ($\frac{l}{D} < \theta$) then the potentials of the particles within the current node can be approximated by a single calculation. In this way the θ value can be used to control the degree of error the algorithm has. This algorithm has been shown to simulate the motion of particle with a reasonable degree of error with a computational complex of $O(N\ log\ N)$ [24, 25].

Barnes-Hut as a Benchmark While the Barnes-Hut treecode method is a very effective method for reducing the computational complexity of the N-body problem, it does not provide a useful benchmark. Because the algorithm reduces calculations by approximating clusters to particle interactions as a single calculation, the number of acceptable approximations depends on the positions of the particles. Thus the time-taken will depend on the random starting configuration of the particles. This makes the performance results of the simulation circumstantial and dependent on the seed and random number generator used. Many executions with multiple configurations must be used to overcome this problem; however, for a benchmark this will take a long time and be unnecessary.

There are other $O(N\ log\ N)$ N-body algorithms available, most notably the fast multipole method (FMM) [28]. However, these other methods use similar approximation methods and suffer from the same issues as the Barnes-Hut method. The All-Pairs algorithm, although computationally more complex and less often used, provided a more stable and meaningful benchmark. The All-Pairs algorithm will perform the same number of potential calculations regardless of the starting configuration of the particles.

4 Benchmark Implementation in CUDA

The GPGPU library chosen for implementing this benchmark is NVidia's CUDA [6]. At the time of publishing, CUDA represents the most complete and easy-to-use GPGPU library. The use of CUDA restricts this benchmark to comparing NVidia graphics cards only, the idea of implementing two benchmarks (one for NVidia one for ATI) was considered but quickly dismissed as the results from the benchmark would be skewed due to the use of different libraries. There may be the potential for a multi-vendor benchmark in the future with the intended releases of libraries such as OpenCL [10] however such libraries are currently in the inception stage. Although CUDA is restricted to NVidia cards, it represents the most suitable library for the purpose of this research.

The benchmark itself will consist of an N-body particle simulation implementing the All-Pairs algorithm with CUDA. To test the performance of the GPU, the simulation will be executed with several system sizes over many time steps. The number of time-steps for each system size will be set such that the time for the benchmark will be several minutes, this is to avoid issues with timing accuracy. The accuracy of the timing will not be increased, the error will simply be insignificant compared to the total time taken for the benchmark.

4.1 Kernel Allocation

To implement this simulation in CUDA, the program must be decomposed into many threads. GPU architectures are best suited to executing large numbers of threads which CUDA organises into blocks (each block can contain a maximum of 512 threads). CUDA executes each block of threads on one multiprocessor at a time and is capable of scheduling large numbers of blocks (the blocks are organised into a grid). The most simple method for allocating threads is to assign one thread to each particle.

In this implementation, each thread is responsible for calculating the total force on its allocated particle. This requires each kernel to access the data from every other particle to perform these calculations. It is relatively easy to implement this algorithm in CUDA; however, to achieve the best performance the program should make use of the optimised memory made available by CUDA. To design a program that utilises the optimisation available on GPU architectures and thus truly test the performance of the GPU, one must understand the types of memory available.

4.2 CUDA Memory

There are many types of memory available for use in a CUDA program, choosing the memory most suited to the task is critical for the performance of the program [13]. The types of memory and the specifics of their usage is described in detail in the CUDA programming guide [6] and in review articles such as [13]. We present only a brief description of the memory types used for input, output and caching within kernels. It is important that this benchmark makes use of these or else the benchmark will measure the performance of GPUs for simulations that ignore the performance-critical optimisations available.

Global Memory is the main GPU memory with the largest size but also slowest access time. A typical Global memory access takes 200-400 ms [6]. CUDA can improve the time required for multiple global memory transactions using a process known as a coalescing. When a half-warp (see the CUDA Programming Guide [6]) of threads (16 sequential threads) access 16 sequential addresses in global memory, CUDA can combine them into one single transaction.

Shared Memory is visible only at the block level. Threads in the same block and read and write to this memory to share data. This type of memory is much faster than global memory but cannot be used to store the values of the field as it cannot be accessed by the host code. However, it can be used to effectively share data to reduce the number of global memory transactions required.

Texture Memory is a memory cache rather than a separate piece of memory. When a value is read from texture memory, the cache will first be checked to determine if the value is already in the cache. If it is not then the texture cache will be reloaded with the value and the values spatially surrounding it from global memory. Texture memory is optimised to cache values in a two-dimensional spatiality, although CUDA does support three-dimensional textures. Texture memory can greatly increase the performance of a program if the threads in each block access values within the spatial size of the texture cache.

Constant Memory is another cache similar to the texture cache. Instead of caching several values in a spatial locality, it is designed for reading constants from memory when all threads in a block read the same value. If a value is stored in the constant cache, all the threads in a half-warp can read from it simultaneously.

Correctly using these memory types can provide a significant speed increase and will test the performance of the GPU when operating in conditions similar to common simulations. To utilise these memory types, the benchmark makes use of a tile calculation method as described in Fast N-Body Simulation with CUDA [29].

Tile Calculation Tile calculation makes use of shared memory to increase the performance of the simulation. The threads are allocated into blocks of p particles with a total of N/p blocks. Each block will process one tile of N/p particles at a time using shared memory to reduce global memory transactions. Each thread calculates the force on its particle by performing the following algorithm:

1. Load one particle's data into shared memory.
2. Synchronise with the other threads in the block.
3. Calculate the force each of the particles stored in shared memory exerts on the thread's particle.
4. Sychronise with the other threads in the block.
5. Repeat 1-4 until all particles have been processed.
6. From the force on the particle, calculate the total change in position and velocity.
7. Write the results back to the output memory.

This the basic algorithm presented by Nyland Et. Al. in "Fast N-Body Simulation with CUDA" [29] and the code for the benchmark has been adapted from their code. One important modification made for this benchmark is the change of integration method. The original integration method used is the leapfrog-Verlet method. While this method is useful, most modern scientific simulations require a higher-order, more stable integration methods to produce meaningful results. The benchmarking version makes use of the Runge-Kutta 2^{nd} order integration method. Listing: 1 is an code snippet from the benchmark that show how the benchmark performs this tile calculation.

Listing 1: Pseudo code that performs the tile calculation N-body. Adapted from the code presented by Nyland Et. Al. in [29]. Note that gravity(float3 acc, float4 p1, float4 p2) is a function that calculates the total acceleration on p2 from p1 and adds it onto acc.

```
#define WRAP(x,m) (((x)<m)?(x):(x-m))
#define SX(i) sharedPos[i]

__global__ void integrateBodies(float4* newPos,
    float4* newVel, float4* oldPos, float4* oldVel,
    float4* oldPos2, float4* oldVel2, float deltaTime,
    int N, bool secondStep) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;
    float4 pos = oldPos[index];

    float3 accel = {0.0f, 0.0f, 0.0f};

    //Go through all tiles
    for(int i = 0; i < N/blockDim.x; i++) {
        // Load one particle's data into share memory
        sharedPos[threadIdx.x] = oldPos[
            WRAP(blockIdx.x+i, gridDim.x) * blockDim.x +
            threadIdx.x];
        // synchronise
        __syncthreads();
        // Calculate total acceleration from the
        // particles in shared memory
        for (unsigned int n=0; n<blockDim.x;) {
            accel = gravity(accel, SX(n++), pos);
            accel = gravity(accel, SX(n++), pos);
            accel = gravity(accel, SX(n++), pos);
            accel = gravity(accel, SX(n++), pos);
        }
        // synchronise
        __syncthreads();
    }

    float4 vel = oldVel[index];

    if(secondStep!=true) {
        // Take first RK2 half step

        pos.x += vel.x * deltaTime * 0.5;
        pos.y += vel.y * deltaTime * 0.5;
        pos.z += vel.z * deltaTime * 0.5;

        vel.x = vel.x + accel.x * deltaTime * 0.5;
        vel.y = vel.y + accel.y * deltaTime * 0.5;
        vel.z = vel.z + accel.z * deltaTime * 0.5;

        // store new position and velocity
        newPos[index] = pos;
        newVel[index] = vel;
    } else {
        // Take second RK2 full step
        float4 pos2 = oldPos2[index];
        float4 vel2 = oldVel2[index];

        pos.x  = pos2.x + vel.x * deltaTime;
        pos.y  = pos2.y + vel.y * deltaTime;
        pos.z  = pos2.z + vel.z * deltaTime;

        vel.x = vel2.x + accel.x * deltaTime;
        vel.y = vel2.y + accel.y * deltaTime;
        vel.z = vel2.z + accel.z * deltaTime;

        // store new position and velocity
        newPos[index] = pos;
        newVel[index] = vel;
    }
}
```

5 Results

The benchmark is set to simulate 1000 time-steps for system sizes of N = [1024, 2048, 4096, 8192], 500 time-steps for N=16,384 and 250 time-steps for N=32,768. The time taken to complete the simulation for each size is recorded and combined together to give a benchmark score. This benchmark will calculate two benchmark scores, the first is the sum of the number of interactions calculated over the time taken to complete the simulations and the second is this first score corrected by the clock speed (C) and number of processors (P) of the GPUs. The second score is intended to represent how efficient the GPU is with its resources. The calculation of these scores can be seen in the following equations:

$$\text{score} = \sum_{N=1024}^{32768} \frac{N^2}{t(N)} \quad (2)$$

$$\text{efficiency} = \frac{1}{P} \times \frac{500}{C} \times \sum_{N=1024}^{32768} \frac{N^2}{t(N)} \quad (3)$$

To test the appropriateness of this benchmark, it has been executed on several GPU architectures to test how well they perform for use as scientific simulations. GPUs from the three generations of CUDA-enabled GPUs have been tested, this is to ensure that the benchmark is applicable to the all recent generations of NVidia GPUs. Included in the list of GPUs are the GeForce 9600M GT and 9400M mobile graphics cards. The score for each GPU along with their total cores and can be seen in Table: 1 which compares the GPUs benchmarked and their final scores.

Because the individual time for the system size is calculated with this benchmark, they can be represented graphically (see Figure: 3). These results are only measured up to a system size of 32,768. Unfortunately system sizes larger than this caused issues with operating system display driver

Table 1: A table of GPUs tested and their final benchmark and efficiency scores.

GPU	Total Stream Processors	Benchmark Score	Efficiency Score
GTX260+	216	7373	25.6
9600M GT	32	1867	58.3
9400GT	16	1245	70.7
9400M	16	970	67.4
8800GTS	96	4637	48.3

time-outs. While this can be overcome by invoking the program multiple times to compute the interactions of only a part of the system at a time, it would provide no more useful information. However, a system size of N=32,768 is sufficient for the purpose of this benchmark and still provides a useful performance measure.

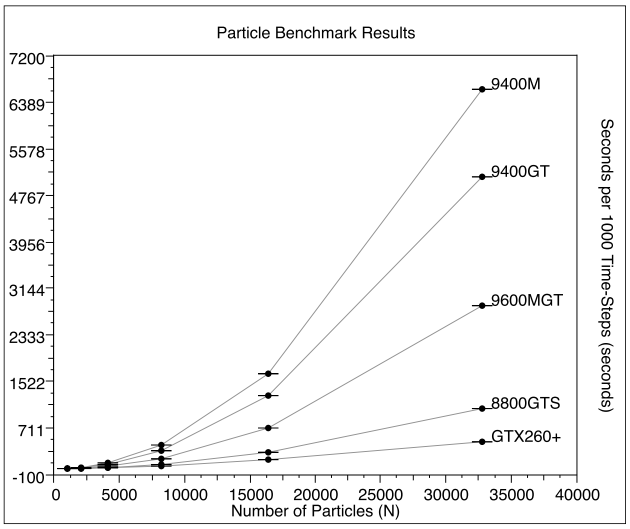

Figure 3: A comparison of the time taken by the GPUs to complete the simulation for system sizes of N = [1024, 2048, 4096, 8192, 16384, 32768].

6 Discussion

Interestingly, yet perhaps not unexpectedly, the higher end graphics cards that provide the best overall performance also make the least efficient use of their resources. It is expected that a GPU with more cores will be less efficient as it has must perform more resource management than a card with fewer cores, however the GTX260+ had under half the efficiency of the GeForce 8000 and 9000 series GPUs.

Interestingly it was the oldest generation graphics card tested (the 8800 GTS) had the best mix of performance and efficiency. While clearly slower in terms of absolute performance compared to the GTX260+, the 8800 GTS had efficiency values comparable to the GeForce 9000 series cards and was almost twice as efficient as the GTX260+.

7 Summary and Conclusions

The benchmark implemented has shown that a N-body simulation is applicable for evaluating the performance of various GPUs with respect to scientific simulations. The results from the GPUs benchmarked using this system provided a comparison in terms of absolute performance as well as efficiency which showed the latest generation graphics cards to be the fastest overall yet also the most inefficient.

The current benchmark implementation utilised NVidia's CUDA API and is thus limited to NVidia graphics cards. We are optimistic about the planned release of libraries such as OpenCL [10] which would allow the benchmark to be extended to compare not only NVidia and ATI graphics cards but also any multi-core architecture.

We believe that the holistic approach of formulating applications-oriented benchmark scores based upon kernel algorithms may be generally applicable and useful when comparing very different device architectures.

References

[1] Playne, D.P., Gerdelan, A.P., Leist, A., Scogings, C.J., Hawick, K.A.: Simulation modelling and visualisation: Toolkits for building artificial worlds. Research Letters in the Information and Mathematical Sciences **12** (2008) 25–50

[2] Eberly, D.H., Shoemake, K.: Game Physics. Number ISBN 978-1558607408. Morgan Kaufmann (2003)

[3] Eberly, D.H.: 3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics. Number ISBN: 978-0122290633. Morgan Kaufmann (2006)

[4] Lennard-Jones, J.: Cohesion. Proc. Royal Soc. **43** (1931) 461–482

[5] Fatahalian, K., Houston, M.: A Closer Look at GPUs. Communications of the ACM **51** (2008) 50–57

[6] NVIDIA® Corporation: CUDA^{TM} 2.0 Programming Guide. (2008) Last accessed November 2008.

[7] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: stream computing on graphics hardware. ACM Trans. Graph. **23** (2004) 777–786

[8] AMD: ATI CTM Guide. (2006)

[9] McCool, M., Toit, S.D.: Metaprogramming GPUs with Sh. A K Peters, Ltd. (2004)

[10] Khronos Group: OpenCL (2008)

[11] Tomov, S., McGuigan, M., Bennett, R., Smith, G., Spiletic, J.: Benchmarking and implementation of probability-based simulations on programmable graphics cards. Computers & Graphics **29** (2005) 71 – 80

[12] Perumalla, K.S., Aaby, B.G.: Data parallel execution challenges and runtime performance of agent simulations on gpus. In: SpringSim '08: Proceedings of the 2008 Spring simulation multiconference, New York, NY, USA, ACM (2008) 116–123

[13] Leist, A., Playne, D., Hawick, K.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Technical Report CSTN-065, Massey University (2008)

[14] Hawick, K.A., James, H.A., Scogings, C.J.: 64-bit architectures and compute clusters for high performance simulations. Technical report, Information and Mathematical Sciences, Massey University, Albany, North Shore 102-904, Auckland, New Zealand (2006)

[15] Hill, M., Marty, M.: Amdahl's law in the multicore era. Computer **41** (2008) 33–38

[16] Yau, H.W., Fox, G.C., Hawick, K.A.: Evaluation of High Performance Fortran through applications kernels. In: Proc. High Performance Computing and Networking 1997. (1997)

[17] Bailey, D., Barscz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: The nas parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, USA. (1994)

[18] Moore, S.K.: Multicore is bad news for supercomputers. IEEE Spectrum **45** (2008) 11

[19] Oskin, M.: The revolution inside the box. Communications of the ACM **51** (2008) 70–78

[20] Sutter, H., Larus, J.: Software and the concurrency revolution. Queue **3** (2005) 54–62

[21] Cantrill, B.: Hidden in plain sight. System Performance **4** (2006) 26–36

[22] Grove, D.A., Coddington, P.D.: Communication benchmarking and performance modelling of mpi programs on cluster computers. J. Supercomput. **34** (2005) 201–217

[23] Newton, I.: Philosophiae Naturalis Principia Mathematica. apud Sa. Smith, London (1687)

[24] Barnes, J., Hut, P.: A hierarchical o(n log n) force-calculation algorithm. Nature **324** (1986) 446–449

[25] Warren, M.S., Salmon, J.K.: A parallel hashed oct-tree n-body algorithm. In: Supercomputing. (1993) 12–21

[26] Anderson, R.: Tree data structures for n-body simulation. In: Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on. (14-16 Oct 1996) 224–233

[27] Appel, A.W.: An Investigation of Galaxy Clustering Using an Asymptotically Fast N-Body Algorithm. PhD thesis, Princeton University (1981)

[28] Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. Journal of Computational Physics **135** (1997) 280–292

[29] Nyland, L., Harris, M., Prins, J.: Fast n-body simulation with cuda. In Nguyen, H., ed.: GPU Gems 3. Addison Wesley Professional (2007)