Computational Science Technical Note **CSTN-120**

Numerical Precision and Benchmarking Very-High-Order Integration of Particle Dynamics on GPU Accelerators

K. A. Hawick and D. P. Playne and M. G. B. Johnson

2011

GPUs offer a powerful acceleration platform for many scientific applications. Numerical integration of classical Newtonian dynamical particles often requires very high-order numerical accuracy. We assess the floating-point precision and performance of various GPUs for applications involving high-order time-step integration methods for particle model simulations using N-squared interactions. We demonstrate how high-order algorithms can be expressed in Compute Unified Device Architecture (CUDA) and present some detailed benchmark data. We show the high numerical power of high-order integration methods such as Hairer's 10^{th} order method and relate its performance to its precision requirements.

Keywords: time-stepping; numerical precision; GPU; CUDA; high-order integration

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-120,
        author = {K. A. Hawick and D. P. Playne and M. G. B. Johnson},
        title = {Numerical Precision and Benchmarking Very-High-Order Integration
                of Particle Dynamics on GPU Accelerators},
        booktitle = {Proc. International Conference on Computer Design (CDES'11)},
        year = {2011},
        number = {CDE4469},
        pages = {83-89},
        address = {Las Vegas, USA},
        month = {18-21 July},
        publisher = {CSREA},
        institution = {Computer Science, Massey University},
        keywords = {time-stepping; numerical precision; GPU; CUDA; high-order integration},
        owner = {kahawick},
        timestamp = {2011.05.16}
}
```

Technical Report CSTN-120

Numerical Precision and Benchmarking Very-High-Order Integration of Particle Dynamics on GPU Accelerators

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

ABSTRACT
GPUs offer a powerful acceleration platform for many scientific applications. Numerical integration of classical Newtonian dynamical particles often requires very high-order numerical accuracy. We assess the floating-point precision and performance of various GPUs for applications involving high-order time-step integration methods for particle model simulations using N-squared interactions. We demonstrate how high-order algorithms can be expressed in Compute Unified Device Architecture (CUDA) and present some detailed benchmark data. We show the high numerical power of high-order integration methods such as Hairer's 10^{th} order method and relate its performance to its precision requirements.

KEY WORDS
time-stepping; numerical precision; GPU; CUDA; high-order integration.

I. Introduction

Graphical Processing Units(GPUs) have risen to great prominence recently with their deployment in many of the leading supercomputers in the Top500 worldwide list. GPUs offer a very-many core solution that is particularly effective at SIMD structured calculations such as field models and other problems where there is a great deal of separable parallelism available in the problem.

In this paper we build on ideas put forward in a previous work[1], on using N-body particle dynamics as a benchmark application problem for accelerator devices such as GPUs. A key open issue for GPUs and related devices is the extent to which they can support double (or even higher) precision floating point calculations. GPUs have proven excellent in data-parallelising integer and 32-bit float problems, and indeed many of the current and forthcoming generation devices do indeed support 64-bit floating point[2]. This support does vary however and in some cases the double precision floating point units are in fact shared across a group of compute cores within the GPU.

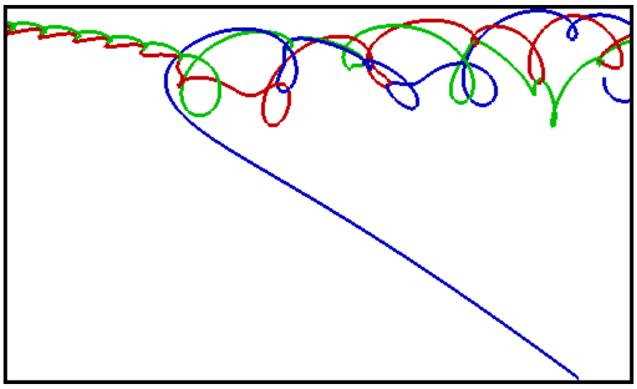
Fig. 1. The motion trails of three interacting particles. Numerically integrated using the Hairer 10^{th} order integration method.

Many problems and simulation applications do not need double precision but some important ones quite definitely must be executed with the highest precision available. The case we discuss in this paper is that of high-order numerical integration in time of the Newtonian equations of motion of classical dynamics for rigid bodies[3]. Many time-integration problems can make do with standard "work-horse" algorithms such as the well known fourth-order Runge-Kutta method[4] and there are several good coded implementations of these widely available[5]. Increasingly modern applications make use of fifth and sixth order algorithms such as those of Dormand and Prince[6], and indeed routines for these algorithms are now widely available in libraries and even in tools such as Matlab[7]. However for precise orbital trajectories or indeed just for cases where conservation of energy is important, even these algorithms are insufficient. Unfortunately, from a mathematical perspective beyond fourth and fifth order the algorithms become significantly more complex and do not scale linearly in the number of floating point operations with the order and precision required.

We have implemented a tenth-order integration method described by Hairer[8] and show that it is very necessary to conserve energy and momentum and so forth in N-body planetary orbital trajectories as might be used in planning spacecraft movements or in simulating a solar system[9], or other astronomical[10] or molecular particle dynamical problems[11], [12]. We show how a sophisticated algorithm like this can be implemented on GPU and use this as a benchmark to compare a number of GPU models with different core and memory combinations. This order and its precision requirements really need a minimum of 64 bit floating point precision to avoid truncation errors and this is therefore a good application driver to explore double and precision floating point requirements for accelerators such as GPUs.

In Section II we discuss the simulation of a N-body particle system and in Section III we discuss some of the low- and high-order integration methods that can be used to integrate their motion over time. In Section IV we show how these simulations and integration methods can be programmed for computation on a GPU. Section V gives some performance and stability results which we then discuss in Section VI. Finally we offer some conclusions and discuss future work in Section VII.

II. N-Body Particle Application

We consider a set of N particles, labelled $i = 0, 1, 2, ..., N-1$ that interact via pair-wise interactions that are dependent on various properties of the particles and most importantly their individual masses. For the purposes of this present paper we consider central forces that depend solely upon the relative distance between particles i and j. For example the Newtonian gravitational potential arising on the i'th particle from the j'th, $V(r_{i,j}$, can be written as:

$$\mathcal{V}(r_{i,j}) = -\frac{Gm_i m_j}{r_{i,j}} \tag{1}$$

The classical (Newtonian mechanical) force can then be written as the gradient of the potential:

$$F = \nabla \mathcal{V}(r) \tag{2}$$

For centralised forces like gravitational systems, we can simply sum pair-wise forces along a vector connecting the particle centres, and for a single such axis the gradient is just a single-variable derivative and hence:

$$F_i = \sum_j G m_i m_j r_{i,j} \tag{3}$$

Given Newton's third law: $F_i = m_i a_i => a_i = F_i/m_i$, we can employ the separate x, y, z components of the acceleration in the Newtonian classical rigid body equations of motion so that we compute changes in particle i's velocity and position.

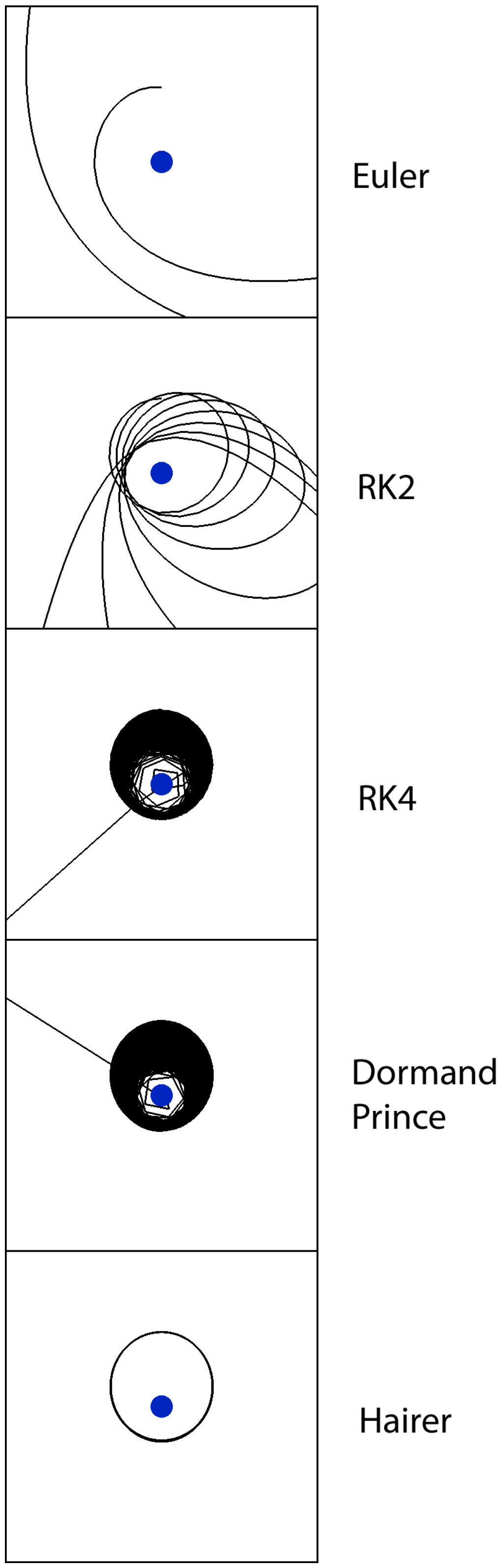

Fig. 2. Numerically integrated particle trace as it orbits a single massive particle. Integration methods from top to bottom are: Euler, RK2, RK4, Dormand-Prince 5th and Hairer 10th. Only the Hairer method remains stable with this time-step of 0.1 for N=2.

For the purposes of the benchmarking application we use here, we simplify to a planar solar system so there is effectively only x, y and no z. A useful benchmark problem is obtained of we model a large central mass with a distribution of very much smaller planetary masses arranged to have close to stable orbital velocities for a randomly chosen distribution of individual masses and initial positions.

So choosing the m_i and $\mathbf{r}_i$ randomly we arrange that for each particle, its kinetic energy ($\frac{1}{2}m_i\mathbf{v}^2$) is initially set equal to its potential energy $\mathcal{V}$ - in the frame of reference of the large-massed central star (indexed by $i = 0$), so that for the remaining planetoid particles we initialise:

$$\mathbf{v}_i = \sqrt{(Gm_0/|\mathbf{r}_i|)} \tag{4}$$

and we decompose $\mathbf{v}_i$ into its v_x, v_y components by arbitrarily choosing a direction of rotation around the star for each particle. We thus have a near stable solar system that is very sensitive to the order and precision supported by the time integration algorithm and hardware.

In the limit of infinite precision we should be able to time-integrate the trajectories in near perfect (energy conserving) orbital ellipses around the central star, and even the small perturbations arising from planet-planet interactions should not give rise to large errors.

In practice of course the numerical integrations routes are very much limited and we can illustrate the effect of different algorithms of increasing order (and computational cost) both visually with plotted trajectory snapshots and also with time plots of the total energy of the system. A good algorithm will conserve energy or at worst exhibit small non-systemic fluctuations. A poor algorithm will exhibit systemic drifts in energy and will fail to conserve it even over short timescales.

III. Numerical Integration

Numerical integration[13] of ordinary differential equations or partial differential equations is a well studied problem with a range of good mathematical algorithm families to choose from. Generally speaking, for the sort of particle trajectory problem we discuss in this paper, higher order methods[14], [15] are better and necessary for systems that have large numbers of particles that interact with inverse square or similar force laws. High-order methods[16] will generally yield better numerical precision if the floating point hardware is available to perform the arithmetic appropriately. The main tradeoff is that higher order methods generally require more intermediate force function calculations and so the computational cost scales considerably worse than linearly with order. Until the widespread availability of accelerator devices such as GPUs the increasingly prohibitive cost of high-order would likely tip the tradeoff balance in favour of methods such as the standard fourth order Runge-Kutta method (RK4) or at best a fifth-order method such as Dormand and Prince (DP5). It is not always trivial to generate software for even higher order methods and still maintain correctness. We have implemented a 10-th order RK method due to Hairer[8].

For the work reported in this paper we employ three-dimensional x,y,z coordinate system even though most test cases consider planar particle systems.

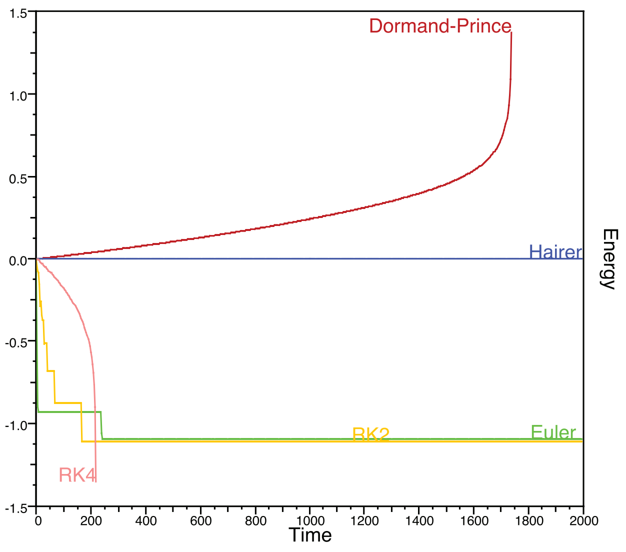

Fig. 3. A plot of the energy of the system shown in Figure 1. The Hairer integration method is stable while the other methods introduce a large degree of error.

Generally explicit RK methods[17] as we discuss here can be expressed as Butcher Tableaux of coefficients: $a_{i,j}; b_i; c_i = \sum_{j=1}^{s} a_{i,j}$. We construct an s-stage method in terms of approximates Y_i with derivative F_i with $F_i = f(Y_i)$ where function f defines our differential equation: $y' = f(y(x)), y(x_0) = y_0$ and for an explicit fixed step h we simply take $x_1 = x_0 + h$. Butcher and other textbooks show that for an s-stage method we derive:

$$y_1 = y_0 + h\sum_{i=1}^{s} b_i F_i \tag{5}$$

with:

$$Y_i = y_0 + h\sum_{j=1}^{s} a_{i,j}F_j, i = 1, 2, ..., s \tag{6}$$

The work of course involves solving these and obtaining useable coefficients which are usually expressed in the form of a Butcher tableau.

Table I shows the tableaux of coefficients for a generalised explicit RK method. Hairer's paper gives the coefficients in full so we do not reproduce them here, but simply note that for a 10'th order accurate method we need $s = 17$ stages and the coefficients need to be expressed to the full precision supportable for the floating point data type.

TABLE I
GENERAL FORM OF A TABLEAU REPRESENTING AN EXPLICIT INTEGRATION METHOD.

$$
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}
$$

1) Initialise particle positions and velocities
2) Synchronise with the other thread
3) Compute sum of all forces on each particle and thus compute accelerations that determine the first differential equation
4) Integrate this to obtain velocities
5) Integrate the other differential equation to obtain distances moved
6) Update all particles accordingly
7) Repeat

The number of force evaluations for different algorithms will constitute the greatest computational cost and for the algorithms we employ are:

TABLE II
INTEGRATION METHODS, THEIR ORDER OF ACCURACY AND NUMBER OF STAGES REQUIRED TO COMPUTE THEM.

Name	Accuracy	Stages	Force Function Evaluations
Euler	1^{st} Order	1-Stage	1(1)
RK2	2^{th} Order	2-Stage	2(2)
RK4	4^{th} Order	4-Stage	4(7)
DP5	5^{th} Order	7-Stage	7(26)
Hairer10	10^{th} Order	17-Stage	17(99)

Table II emphasizes the difference in the order of accuracy for each integration method and the number of stages required to compute them. By saving the force evaluations in memory the number of force evaluations can be made equal to the number of stages. The number in brackets shows the number of force evaluations required if they are not stored to reduce memory use.

It is thus seen that a performance accelerator such as a many-cored GPU is important to offset the increasing operation count needed to obtain high numerical accuracy for the particle trajectories.

IV. GRAPHICAL PROCESSING UNITS

Graphical Processing Units or GPUs have steadily increasing in popularity for computing scientific simulations [18]. The high computational throughput and relatively low cost of GPUs have made them an attractive platform for parallel processing. This is known as General Purpose computation on GPUs or GPGPU and there are a number of APIs available. The most popular and powerful GPGPU API is NVidia's CUDA [19] which only works on the NVidia GeForce and Tesla graphics cards, this is the API used for this research.

Two generations of NVidia GPUs are considered in this work the Tesla and Fermi architectures. Both of these GPUs contain many (up to 512) cores known as scalar processors which are organised into multiprocessors. Tesla multiprocessors contain 8 SPs which multiprocessors on Fermi GPUs each contain 32. Programs can be run on these cores by splitting a problem into a large number of threads which are managed and scheduled by the GPU hardware. These threads are organised into blocks, each block of threads will be processed by one multiprocessor.

GPUs can manage this many cores by dropping the cache hierarchy common in most CPU architectures, eliminating the cache coherency problem that limits the expansion of cores in CPUs. To replace this cache, GPUs contain a number of specific memory types which can be used explicitly by developers to cache memory access. These memory types are:

- **Global** memory - the main memory of the GPU device. Can be accessed by the CPU host as well as any thread executing on any multiprocessor.
- **Shared** memory - is shared between the threads of a multiprocessor. It can be accessed very quickly by all the threads executing in the same block.
- **Texture** memory - is a cached access for global memory. Designed for threads in the same block accessing values from global memory in the same spatial locality.
- **Constant** memory - is another cached method of accessing global memory. Designed to allow all threads in a block to read the same value from global memory at the same time.

The new generation Fermi architecture GPUs have also introduced a L1/L2 cache back into the design. This cache only guarantees that data written to the cache will be written to global memory once the block has finished executing. This way the problem of cache coherency is avoided.

The Tesla architecture GT200 GPUs and all of the Fermi GPUs support double precision floating point calculations, however there is still a performance loss associated with it. The higher-order integration methods discussed in Section III are pointless without the use of double precision floating point. We aim to compare the performance of different GPUs with respect to computing high-order integration methods using double precision.

A. CUDA N-Body Implementation

The CUDA implementation of the particle dynamics simulator uses the all-pairs tile calculation method described in "Fast N-Body Simulation with CUDA" [20], [1]. This method uses shared memory to cache particles and reduce global memory access. Each thread block will load a tile of particles into the shared cache, process it and then load the next tile. To process N particles with tiles of size T this process must be performed $\frac{N}{T}$ times.

This method is used to compute the acceleration of each particle due to gravity. To implement the higher order integration methods, the CUDA kernel has been adapted to use an array of coefficients (taken from the butcher tableau of the integration method) to compute the next state. The kernel must be called s times to compute the intermediate and final system according to the integration method. A set of s system states are computed which are then used in the final computation. Each state is computed as the sum of the derivatives of the previous saved states multiplied by the coefficients of the table. This must be performed for both the position and velocity of the particles.

The CUDA kernel that computes a single system state is shown in Listing 1. This kernel will compute a single state $s+1$ using pos to store the positions of the particles, vel to store the velocities, acc to store the accelerations, $mass$ to store the masses and *coeff* to store the coefficients.

Listing 1. CUDA kernel to compute a single stage of the integration method defined by `coeff`.

```
_shared_ double4 cache[BLOCK_SIZE];
_global_ void kernel(double4 **pos, double4 **vel,
                     double4 **acc, double **coeff,
                     double *mass,   int s) {
  int i=blockIdx.x*blockDim.x+threadIdx.x;
   //Load particle i from step s
  double4 p_i = pos[s][i];
  double4 a_i = make_double4(0.0,0.0,0.0,0.0);
  //Go through all tiles
  for(int t = 0; t < N/blockDim.x; t++) {
    //Calculate tile index
    int ti=blockIdx.x+t;
    ti = (ti < gridDim.x) ? ti : ti-gridDim.x;
    ti = ti*blockDim.x;
    //Load and process tile
    cache[threadIdx.x] = pos[s][bi+threadIdx.x];
    __syncthreads();
    for (int j = 0; j < blockDim.x; j++) {
      double4 pn = cache[j];
      if((ti+j) != i) {
        double d = distance(pi-pn);
        double av = -(G* mass[tile_index+j])/(d*d);
        ai = (av/d) * (pi-pn);
      }
    }
    __syncthreads();
  }
  acc[s][i] = ai;
  double4 pn = pos[0][i];
  double4 vn = vel[0][i];
  for(int c = 0; c <= s; c++) {
    pn += vel[c][i]*coeff[s][c]*h;
    vn += acc[c][i]*coeff[s][c]*h;
  }
  pos[(s+1)][i] = pn;
  vel[(s+1)][i] = vn;
}
```

This kernel can be used to integrate the system for any of the integration methods discussed in Section III.

V. Performance Results

Figure 4 shows the time taken by a GTX580 to compute 1000 steps of the N-body simulation using the different integration methods. The results scale as expected with the Hairer method taking the longest.

Figure 5 shows a comparison between the different NVidia GPUs and their performance for computing the N-Body simulation with the Hairer 10^{th} order integration method. The figure shows the performance for both the single and double precision floating point calculations for reference. The GPUs compared are the Tesla architecture (GTX260 and GTX295) and the Fermi architecture (GTX480, GTX580 and C2070).

As expected all graphics cards showed higher performance processing single- rather than double-precision floating point numbers. The GTX 580 showed the best overall performance for both the single- and double-precision simulations. This is expected as the GTX580 has the highest clock speeds and number of cores. One point of interest is the C2070s double-precision performance as compared to the GTX 480.

The GTX480 performs slightly better than the C2070 when processing single-precision floating point values but the C2070 provides significantly better performance when processing double-precision.

TABLE III
Comparison of the different integration methods to compute a single unit of simulation time with error $e < 1 \times 10^{-12}$.

Method	Time per Simulated Second
Euler	N/A
RK2	0.2292 seconds
RK4	0.0070 seconds
Dormand-Prince	0.0042 seconds
Hairer	0.0010 seconds

Table III gives a comparison of the time taken for the different integration methods to compute a single simulation second for a fixed error. While the higher-order methods are more expensive to compute they are more accurate and therefore capable of numerical stability in problem regimes inaccessible to the lower-order algorithms. It can be seen from the table that the 17-stage Hairer integration method can compute a simulated second the fastest as it can support the largest simulation time-step.

VI. Discussion

The performance curves in Figure 4(right) and Figure 5 (right) both show a kink at $N = 1024 \cdots 4096$ depending on the GPU model. This is attributable to the the limited

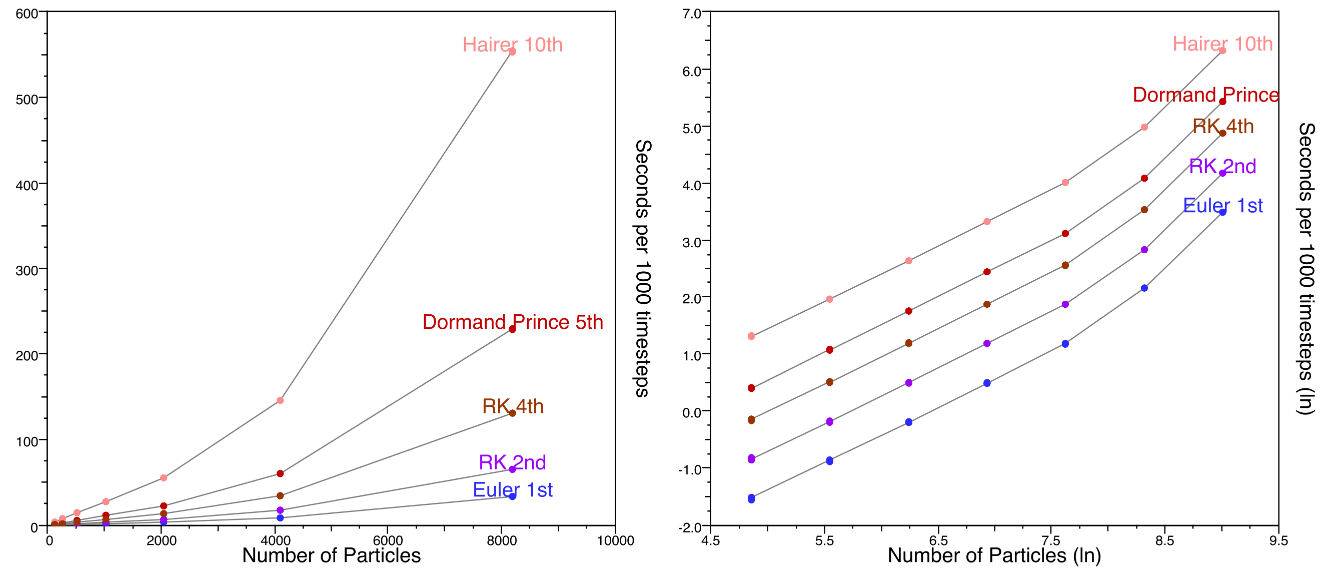

Fig. 4. The performance results of the Fermi architecture GTX580 for processing the N-body simulation with different integration methods.

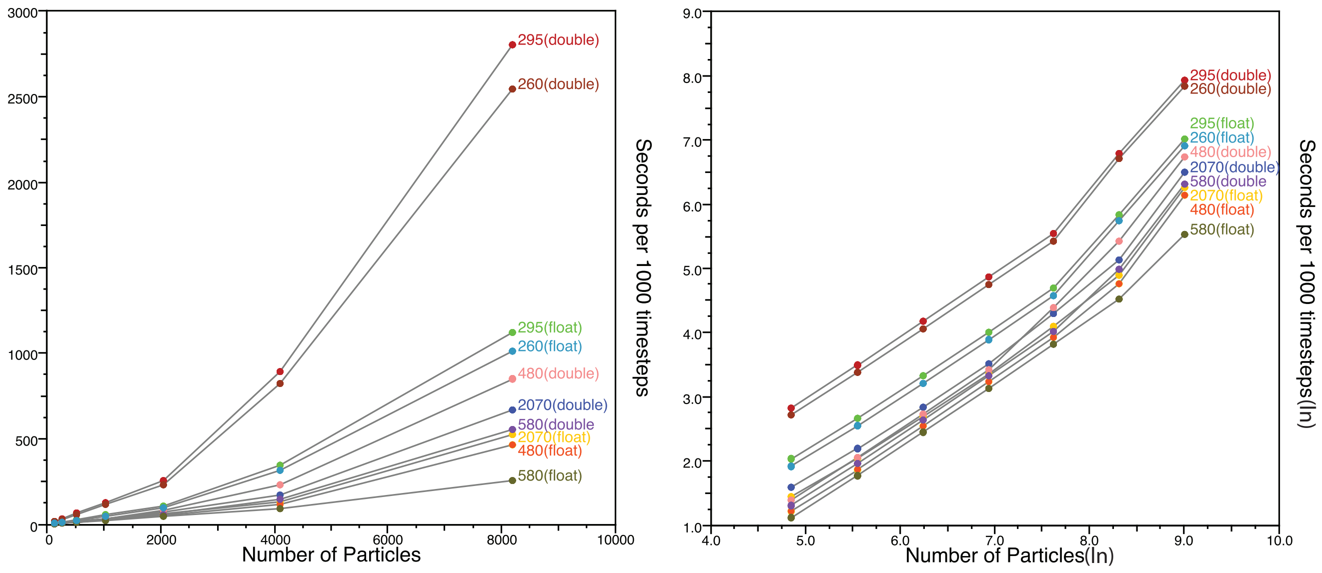

Fig. 5. A comparison of different GPUs computing the N-Body simulation using Hairer 10^{th} order integration method. This plot shows the performance for the GTX260, GTX295, GTX480, GTX 580 and Tesla C2070. Timings for both double and single precision floating point calculations are shown.

processors of the GPU, above this transition multiple thread block must be executed on a single multiprocessor. Systems below this size cannot utilise the all of the GPU multiprocessors and thus do not make optimal use of the GPU hardware. For the purposes of benchmarking the system must be sufficiently large to fully utilise all available multiprocessors. The optimal point benchmarking system is at the position of this kink, however this not constant for all GPU models.

The benchmark described in [1] describes a general purpose application with low precision integration methods and floating point numbers. Whereas, this paper is aimed at specific scientific computing applications where the precision of the integration methods and double precision floating point numbers are required. Using these methods it allows us to exploit the differences between different generations and data types of modern NVidia GPU's.

VII. Conclusions

We have shown that high-order time integration algorithms for N-Body classical particle dynamics make an interesting and challenging benchmark application for computational performance accelerators such as GPUs. We have explored the computational load of methods ranging from the simple Euler single stage method, through common workhorse algorithms such as fourth-order Runge-Kutta and emerging algorithms such as the Dormand and Prince fifth order method. We have also implemented the relatively unknown, superior but computationally expensive 10th order Hairer algorithm and shown the increasing number force-functions used by such seventeen-stage methods. Nevertheless we have confirmed that such numerically superior methods do support problem configurations with high curvature of variable values, that are simply not feasible with lower-order methods. We have also explored the notion of simulated seconds per second and have shown that there are problem regimes where it is actually faster in terms of the number of simulated seconds per clock second to use a high order method such as Hairer's.

We have explored properties of the NVidia GPUs and have found it quite feasible to implement these codes using CUDA although we note the increasing code complexity required in making a correct implementation. The 10th order method uses the full numerical precision available in 64-bit arithmetic and it therefore represents a threshold in utility. An even higher order method would perhaps necessarily be crippled if only run with 64 bit floating point precision.

We have confirmed the float and double performance of NVidia's high-end GPUs and reaffirmed the ongoing importance of double precision accelerator units for use of massively multi-core data parallel devices such as GPUs for these sort of high accuracy N-Body calculations.

There is further scope for using these algorithms on GPU-enabled clusters rather than just on a single GPU-accelerated CPU node as we have discussed in this paper. There is also scope to explore other numerical integration algorithms such as the family of adaptive step-size explicit Runge-Kutta methods or implicit methods[17]. It is likely that this class of problem will remain an important benchmark for floating-point accelerators.

References

[1] Playne, D.P., Johnson, M.G.B., Hawick, K.A.: Benchmarking GPU Devices with N-Body Simulations. In: Proc. 2009 International Conference on Computer Design (CDES 09) July, Las Vegas, USA. Number CSTN-077 (2009)

[2] Knuth, D.: The Art of Computer Programming: Seminumerical Algorithms. 3rd edn. Volume 2. Addison-Wesley (1997)

[3] Hairer, E., Vilmart, G.: Preprocessed discrete MoserVeselov algorithm for the full dynamics of a rigid body. J. Phys. A: Math. and Gen. **39** (2006) 13225

[4] Hairer, E., Wanner, G.: Algebraically Stable and Implementable Runge-Kutta Methods of High Order. SIAM J. Numer. Anal. **18** (1981) 1098–1108

[5] Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C. 2nd edn. Cambridge University Press (1992) ISBN 0-521-43108-5.

[6] Dormand, J., Prince, P.J.: A Family of Embedded Runge-Kutta Formulae. J. of Computational and Applied Maths. **6** (1980) 19–26

[7] The MathWorks: Matlab. available at http://www.mathworks.com (2007)

[8] Hairer, E.: A Runge-Kutta Method of Order 10. J. Inst. Maths. Applics. **21** (1978) 47–59

[9] Playne, D.P.: Notes on particle simulation and visualisation. Hons. thesis, Computer Science, Massey University (2008)

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

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

[12] Allen, M., Tildesley, D.: Computer simulation of liquids. Clarendon Press (1987)

[13] Cohen, G.C.: Higher-Order Numerical Methods for Transient Wave Equations. Number ISBN 3-540-41598-X in Scientific Computation. Springer (2002)

[14] Enright, W.H., Higham, D.J., Owren, B., Sharp, P.W.: A Survey of the Explicit Runge-Kutta Method. Technical Report 291/94, Computer Science, University of Toronto (1995)

[15] Shampine, L.F.: Some practical Runge-Kutta Formulas. Mathematics of Computation **46** (1986) 135–150 ISSN: 0025-5718.

[16] Tu, V.P., Tlusty, J.: Higher-order (2,8) fdtd method for solving multiconductor transmission-line equations. In: 2004 Large Engineering Systems Conf. on Power Engineering. (2004)

[17] Butcher, J.C.: Numerical Methods for Ordinary Differential Equations. Second edition edn. Number ISBn 978-0-470-72335-7. Wiley (2008)

[18] Leist, A., Playne, D., Hawick, K.: Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Concurrency and Computation: Practice and Experience **21** (2009) 2400–2437 CSTN-065.

[19] NVIDIA® Corporation: NVIDIA CUDA^{TM} C Programming Guide Version 3.2. (2010) Last accessed December 2010.

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