



Universiteit
Leiden
The Netherlands

The gravitational billion body problem : Het miljard deeltjes probleem Bédorf, J.

Citation

Bédorf, J. (2014, September 2). *The gravitational billion body problem : Het miljard deeltjes probleem*. Retrieved from <https://hdl.handle.net/1887/28464>

Version: Corrected Publisher's Version

License: [Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden](#)

Downloaded from: <https://hdl.handle.net/1887/28464>

Note: To cite this publication please use the final published version (if applicable).

Cover Page



Universiteit Leiden



The handle <http://hdl.handle.net/1887/28464> holds various files of this Leiden University dissertation

Author: Jeroen Bédorf

Title: The gravitational billion body problem / Het miljard deeltjes probleem

Issue Date: 2014-09-02

2 | Sapporo2: A versatile direct N -body library

Astrophysical direct N -body methods have been one of the first production algorithms to be implemented using NVIDIA's CUDA architecture. Now, almost six years later, the GPU is the most used accelerator device in astronomy for simulating stellar systems. In this paper we present the implementation of the **Sapporo2** N -body library, which allows researchers to use the GPU for N -body simulations with little to no effort. The first version, released five years ago, is actively used, but lacks advanced features and versatility in used numerical precision and support for higher order integrators. In this updated version we have rebuilt the code from scratch and added support for **OpenCL**, multi-precision and higher order integrators. We show how to tune these codes for different GPU architectures and present how to continue utilizing the GPU as optimally as possible even when only a small number of particles is integrated. This careful tuning allows **Sapporo2** to be faster than **Sapporo1** even with the added options and double precision data loads. The code shows excellent scaling on a range of NVIDIA and AMD GPUs in single and double precision accuracy.



2.1 Background

The class of algorithms, commonly referred to as direct N -body algorithms is still one of the most commonly used methods for simulations in astrophysics. These algorithms are relative simple in concept, but can be applied to a wide range of problems. From the simulation of few body problems, such as planetary stability to star-clusters and even small scale galaxy simulations. However, these algorithms are also computationally expensive as they scale as $O(N^2)$. This makes the method unsuitable for large N ($> 10^6$), for these large N simulations one usually resorts to a lower precision method like the Barnes-Hut tree-code method Barnes and Hut (1986) that scales as $O(N \log N)$ or the Particle Mesh method that scales as $O(N)$ (e.g. Hohl and Hockney (1969); Hockney and Eastwood (1981)). These methods, although faster, are also notably less accurate and not suitable for simulations that rely on the high accuracy that direct integration offers. On the other end of the spectrum you can find even higher accuracy methods which use arbitrary precision (Boekholt et al. in prep). The results of Boekholt et al. indicate that the accuracy offered by the default (double precision) direct N -body methods is sufficient for most scientific problems.

The direct N -body algorithm is deceptively simple, in the basic form it performs N^2 gravitational computations, which is an embarrassingly parallel problem that can be efficiently implemented on almost any computer architecture with a limited amount of code lines. A number of good examples can be found on the Nbabel.org website. This site contains examples of a simple N -body simulation code implemented in a wide range of programming languages. However, in practice there are many variations of the algorithms in use, with up to 8th order integrations Nitadori and Makino (2008), algorithmic extensions such as block-time stepping McMillan (1986), neighbour-schemes Ahmad and Cohen (1973), see Bédorf and Portegies Zwart (2012) and references therein for more examples. These variations transform the simple $O(N^2)$ shared time-step implementation in one with many dependencies and where the amount of parallelism can differ per time-step. Especially the dynamic block time-stepping method adds complexity to the algorithm, since the number of particles that participate in the computations change with each integration step. This variable number of particles involved in the computations forces the use of different parallelisation strategies. In the worst case there is only one particle integrated which eliminates most of the standard parallelisation methods for N^2 algorithms. There is extensive literature on high performance direct N -body methods with the first being described in 1963 Aarseth (1963). The method has been efficiently implemented on parallel machines McMillan (1986), vector machines Hertz and McMillan (1988) and dedicated hardware such as the GRAPEs Makino and Taiji (1998). For an overview we refer the interested reader to the following reviews Bédorf and Portegies Zwart (2012); Heggie and Hut (2003); Dehnen and Read (2011)

In this paper we present our direct N -body library, Sapporo2. The library contains built-in support for second order leap-frog (GRAPE5), fourth order Hermite (GRAPE6) and the sixth order Hermite integrators. The numerical precision can be specified at run time and depends on requirements for performance and accuracy. Furthermore, the library can keep track of the nearest neighbours by returning a list containing all particles within a



certain radius. Depending on the available hardware the library operates with CUDA and OpenCL, and has the option to run on multiple-GPUs if installed in the same system. The library computes the gravitational force on particles that are integrated with block time-step algorithms. However, the library can trivially be applied to any other N^2 particle method by replacing the force equations.

2.2 Methods

With Graphic Processing Units (GPUs) being readily available in the computational astrophysics community for over 5 years we will defer a full description of their specifics and peculiarities Bédorf and Portegies Zwart (2012); Belleman et al. (2008); Nyland et al. (2007); NVIDIA (2013a). Here we only give a short overview to stage the context for the following sections. In GPU enabled programs we distinguish two parts of code. The ‘host’ code, used to control the GPU, is executed on the CPU; whereas the ‘device’ code, performing the majority of the computations, is executed on the GPU. Each GPU consists of a set of multiprocessors and each of these multiprocessors contains a set of computational units. We send work to the GPU in blocks for further processing by the multiprocessors. In general a GPU requires a large amount of these blocks to saturate the device in order to hide most of the latencies that originate from communication with the off-chip memory. These blocks contain a number of threads that perform computations. These threads are grouped together in ‘warps’ for NVIDIA machines or ‘wavefronts’ on AMD machines. Threads that are grouped together share the same execution path and program counter. The smaller the number of threads that are grouped the smaller the impact of thread divergence. On current devices a warp consists of 32 threads and a wavefront contains 64 threads. This difference in size has an effect on the performance (see Section 2.3).

2.2.1 Parallelisation method

To solve the mutual forces for an N -body system the forces exerted by the j -particles (sources) onto the i -particles (sinks) have to be computed. Depending on the used algorithm the sources and sinks can either belong to the same or a completely different particle set. Neither is it required that these sets have the same dimensions. In worst case situations this algorithm scales as $O(N^2)$, but since each sink particle can be computed independently it is also embarrassingly parallel. The amount of parallelism however depends on the number of sink particles. For example, in high precision gravitational direct N -body algorithms that employ block-time stepping the number of sink particles ranges between 1 and N . In general the number of sinks is \ll than the number of sources, because only the particles of which the position and velocity require an update are integrated McMillan (1986). As a consequence the amount of available parallelism in this algorithm is very diverse, and depends directly on the number of active sink particles.

Currently there are two commonly used methods for solving N^2 like algorithms using GPUs. The first performs parallelisation over the sink particles Hamada and Iitaka (2007); Belleman et al. (2008); Nyland et al. (2007) which launches a separate compute thread for each sink particle. This is efficient when the number of sinks is large ($> 10^4$), because



then the number of compute threads is sufficiently high to saturate the GPU. However, when the number of sink particles is small ($\leq 10^4$) there are not enough active compute threads to hide the memory and instruction latencies. As a result the GPU will be under utilized and only reaches a fraction of the available peak performance. We expect that future devices require an even larger number of running threads to reach peak performance, in which case the number of sink particles has to be even larger to continuously saturate the device. However, adjusting the number of sink particles to keep parallel efficiency is not ideal, because then one artificially increases the amount of work (and runtime) in favor of efficiency. Therefore, a second method was introduced in Sapporo1 Gaburov et al. (2009) which takes a slightly different approach. In Sapporo1 we parallelize over the source particles and keep the number of sink particles that is concurrently integrated fixed to a certain number. The source particles are split into subsets, each of which forms the input against which a set of sink particles is integrated. The smaller the number of sink particles the more subsets of source particles we can make. It is possible to saturate the GPU with enough subsets, so if the combined number of sink and source particles is large enough¹ you can reach high performance even if the number of sinks or sources is small.

Of the two parallelisation methods the first one is most efficient when using a shared-time step algorithm, because fewer steps are involved in computing the gravity. However, Sapporo1 is more suitable for block-time stepping methods commonly used in high precision gravitational N -body methods. Even though this method requires an extra step to combine the partial results from the different subsets. The Sapporo1 method is also applied in this work. With Sapporo1 being around for 5 years we completely rewrote it and renamed it to Sapporo2 which is compatible with current hardware and is easy to tune and adapt to future generation accelerator devices and algorithms. The next set of paragraphs describe the implementation and the choices we made.

2.2.2 Implementation

CUDA and OpenCL

When NVIDIA introduced the CUDA framework in 2007 it came with compilers, runtime libraries and examples. CUDA is an extension to the ‘C’ programming language and as such came with language changes. These extensions are part of the device and, more importantly, part of the host code². The use of these extensions requires that the host code is compiled using the compiler supplied by NVIDIA. With the introduction of the ‘driver API’ this was no longer required. The ‘driver API’ does not require modifications to the ‘C’ language for the host code. However, writing CUDA programs with the ‘driver API’ is more involved than with the ‘runtime API’, since actions that were previously done by the NVIDIA compiler now have to be performed by the programmer.

When the OpenCL programming language was introduced in 2009 it came with a set of extensions to the ‘C’ language to be used in the device code. There are no changes to the language used for writing the host code, instead OpenCL comes with a specification of

¹The exact number required to reach peak performance depends on the used architecture, but if the total number of gravitational interactions is $\geq 10^6$ it is possible to saturate the GPU

²The most notable addition is the ‘<<<>>>’ construction to start compute kernels.



functions to interact with the device. This specification is very similar to the specification used in the CUDA driver API and follows the same program flow.

In order to support both OpenCL and CUDA in Sapporo2 we exploited the similarity between the CUDA driver API and the OpenCL API. We developed a set of C++ classes on top of these APIs which offer an unified interface for the host code. The classes encapsulate a subset of the OpenCL and CUDA functions for creating device contexts, memory buffers (including functions to copy data) and kernel operations (loading, compiling, launching). Then depending on which class is included at compile time the code is executed using OpenCL or CUDA. The classes have no support for the more advanced CUDA features such as OpenGL and Direct3D interoperability.

Kernel-code With the wrapper classes the host-code is language independent. For the device code this is not the case, even though the languages are based on similar principles the support for advanced features like C++ templates, printing and debugging functionality in CUDA makes it much more convenient to develop in pure CUDA. After that we port the working code to OpenCL. The use of templates in particular reduces the amount of code. In the CUDA version all possible kernel combinations are implemented using a single file with templates. For OpenCL a separate file has to be written for each combination of integrator and numerical precision.

The method used to compute the gravitational forces is comparable to the method used in Sapporo1 with only minor changes to allow for double precision data loads/stores and more efficient loop execution.

Numerical Accuracy

During the development of Sapporo1 GPUs lacked support for IEEE-754 double precision computations and therefore all the compute work was done in either single or double-single precision³. The resulting force computation had similar precision as the, at that time, commonly used GRAPE hardware Makino and Taiji (1998); Gaburov et al. (2009). This level of accuracy is sufficient for the fourth order Hermite integration scheme Makino and Aarseth (1992); Portegies Zwart and Boekholt (2014). Currently, however there are integrators that accurately solve the equations of motions of stars around black-holes, planets around stars and similar systems that encounter high mass ratios. For these kind of simulations one often prefers IEEE-754 double precision to solve the equations of motion. The current generation of GPUs supports IEEE-754, which enables computations that require this high level of accuracy. Therefore the data in Sapporo2 is always stored in double precision. The advantage of this is that we can easily add additional higher order integrators that require double precision accuracy computations without having to rewrite major parts of the host code. Examples of such integrators are the 6th and 8th order Hermite integrators Nitadori and Makino (2008). The performance impact of double precision storage on algorithms that do not require double precision computations is limited. Before the actual computations are executed the particle properties are converted to either float or double-single and the precision therefore does not influence the computational performance. The

³In this precision, the number of significant digits is 14 compared to 16 in IEEE double precision



penalty for loading and storing double the amount of data is relative small as can be seen in the result section where Sapporo1 is compared to Sapporo2.

multiple GPUs

Our new N -body library can distribute the computational work over multiple GPUs, as long as they are installed in the same system. While in Sapporo1 this was implemented using the boost threading library, this is now handled using OpenMP. The multi-GPU parallelisation is achieved by parallelisation over the source particles. In Sapporo1 each GPU contained a copy of all source particles (as in Harfst et al. (2007)), but in Sapporo2 the source particles are distributed over the used devices using the round-robin method. Each GPU now only holds a subset of the source particles which reduces memory requirements, transfer time and the time to execute the prediction step on the source particles. However, the order of the particle distribution and therefore the addition order is changed when comparing Sapporo1 and Sapporo2. This in turn can lead to differences in the least significant digit when comparing the computed force of Sapporo1 to Sapporo2.

Other differences

The final difference between Sapporo1 and Sapporo2 is the way the partial results of the parallelisation blocks are combined. Sapporo1 contains two computational kernels to solve the gravitational forces. The first computes the partial forces for the individual blocks of source particles, and the second sums the partial results. With the use of atomic operators these two kernels can be combined, which reduces the complexity of maintaining two compute kernels when adding new functionality at a minimal performance impact. The expectation is that future devices require more active threads to saturate the GPU, but at the same time offer improved atomic performance. The single kernel method that we introduced here will automatically scale to future devices and offers less overhead than launching a separate reduction kernel.

2.3 Results

In astrophysics the current most commonly used integration method is the fourth order Hermite Makino and Aarseth (1992) which requires per particle the nearest neighbour and a list of neighbours within a certain radius. This is what Sapporo1 computes and how the GRAPE hardware operates Makino and Taiji (1998). The used numerical precision in this method is the double-single variant. In order to compare the new implementation with the results of Sapporo1, all results in this section, unless indicated otherwise, refer to the double-single fourth order Hermite integrator.

For the performance tests we used different machines, depending on which GPU was used. All the machines with NVIDIA GPUs have CUDA 5.5 toolkit and drivers installed. For the machine with the AMD card we used toolkit version 2.8.1.0 and driver version 13.4.

The full list of used GPUs can be found in Tab. 2.1, the table shows properties such as clock speed and number of cores. In order to compare the various GPUs we also show



the theoretical performance, relative with respect to the GTX480. Since, theoretical performance is not always reachable we also show the relative practical performance as computed with a simple N -body kernel that is designed for shared-time steps, similar to the N -body example in the CUDA SDK Nyland et al. (2007).

	Cores	Core MHZ	Mem MHZ	Mem bw	TPP	PPP
GTX480	480	1401	3696	384	1	1
GTX680	1536	1006	6008	256	2.3	1.7
K20m	2496	706	5200	320	2.6	1.8
GTX Titan	2688	837	6144	384	3.35	2.2
HD7970	2048	925	5500	384	2.8	2.3

Table 2.1: GPUs used in this work. The first column indicates the GPU, the second column the number of computational cores and the third their clock speed. The fourth and fifth column show the memory clock speed and memory bus width. The sixth and seventh column indicate the relative performance, when using single precision, where we set the performance of the GTX480 to 1. For the sixth column these numbers are determined using the theoretical peak performance (TPP) of the chips. The seventh column indicates the relative practical peak performance (PPP) which is determined using a simple embarrassingly parallel N -body code.

2.3.1 Thread-block configuration

Since Sapporo2 is designed around the concept of a fixed number of blocks and threads (see Section 2.2) the first thing to determine is the optimal configuration of threads and blocks. We test a range of configurations where we vary the number of blocks per multi-processor and the number of threads per block. The results for four different GPU architectures are presented in Fig. 2.1. In this figure each line represents a certain number of blocks per multi-processor, N_{blocks} . The x-axis indicates the number of threads in a thread-block, $N_{threads}$. The range of this axis depends on the hardware. For the HD7970 architecture we can not launch more than $N_{threads} = 256$, and for the GTX480 the limit is $N_{threads} = 576$. For the two Kepler devices 680GTX and K20m we can launch up to $N_{threads} = 1024$ giving these last two devices the largest set of configuration options. The y-axis shows the required wall-clock time to compute the forces using the indicated configuration, the bottom line indicates the most optimal configuration.

For the 680GTX and the K20m the N_{blocks} configurations reach similar performance when $N_{threads} > 512$. This indicates that at that point there are so many active threads per multi-processor that there are not enough resources (registers and/or shared-memory) to accommodate multiple thread-blocks per multi-processor at the same time. To make the code suitable for block time-steps the configuration with the least number of threads that gives the highest performance would be the most ideal. For the HD7970 this is $N_{threads} = 256$ while for the Kepler architectures $N_{threads} = 512$ gives a slightly lower execution time than $N_{threads} = 256$ and $N_{threads} = 1024$. However, we chose to use $N_{threads} = 256$ for all configurations and use 2D thread-blocks on the Kepler devices to launch 512 or 1024 threads. For each architecture the optimal configuration is indicated with the circles

in Fig. 2.1.

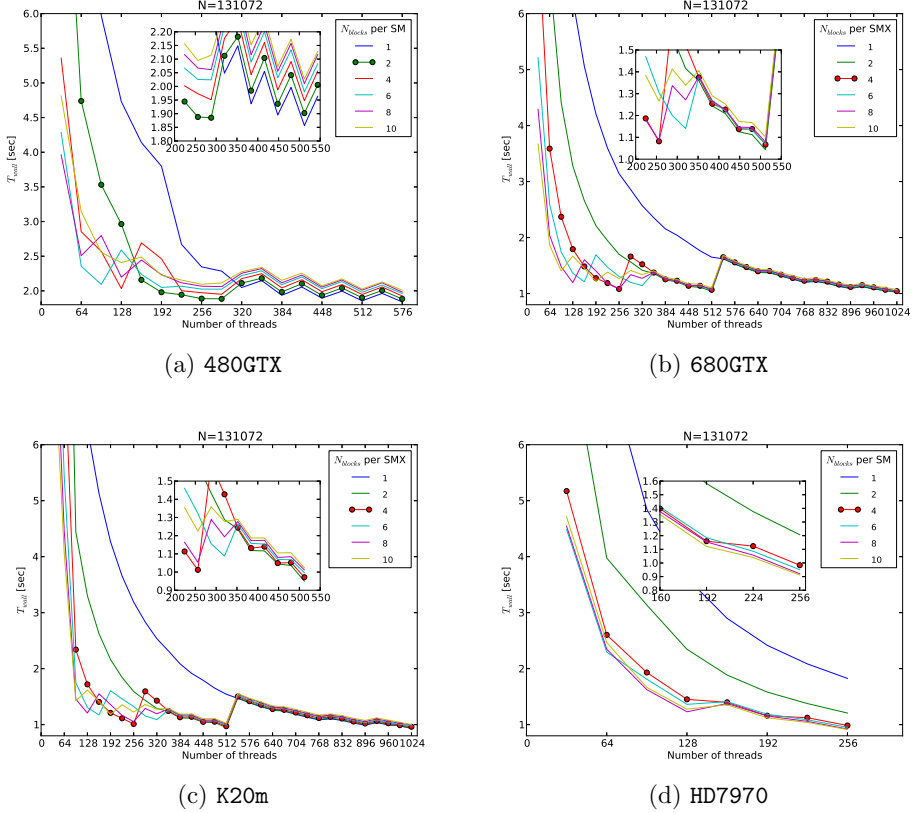


Figure 2.1: The figure shows the required integration time (y-axis) for $N = 131072$ source particles using different number of sink particles (number of threads, x-axis). Each line indicates a different configuration. In each configuration we changed the number of blocks launched per GPU multi-processor for different GPU architectures. Shown in panel A NVIDIA's Fermi architecture, in panel B the NVIDIA Kepler, GK104 architecture in panel C the NVIDIA Kepler, GK110 and the AMD Tahiti architecture in panel D. The AMD architectures are limited to 256 threads. The configurations that we have chosen as our default settings for the number of blocks are the lines with the filled circle markers.

2.3.2 Block-size / active-particles

Now we inspect the performance of Sapporo2 in combination with a block-time step algorithm. We measured the time to compute the gravitational forces using either the NVIDIA GPU Profiler or the built-in event timings of OpenCL. The number of active sink particles, N_{active} , is varied between 1 and the optimal N_{threads} as specified in the

previous paragraph. The results are averaged over 100 runs and presented in Fig. 2.2. We used 131072 source particles which is enough to saturate the GPU and is currently the average number of particles used in direct N -body simulations.

The straight striped lines in Fig. 2.2 indicate the theoretical linear scaling from $(0, 0)$ to $(256, X)$ where X is the execution time of the indicated GPU when $N_{active} = 256$. Visible in the figure are the jumps in the execution time that coincide with the warp (wavefront) size of 32 (64). For NVIDIA devices we can start 2D thread-blocks for all values of N_{active} , since the maximum number of threads that can be active on the device is ≥ 512 . The effect of this is visible in the more responsive execution times of the NVIDIA devices when decreasing N_{active} compared to the AMD device. Each time N_{active} drops below a multiple of the maximum number of active threads, the execution time will also decrease. Up to $N_{active} \lesssim 64$ after which the execution time goes down linearly, because of the multiple blocks that can be started for any value of N_{active} . The lines indicated with ‘1D’ in the legend show the execution time if we would not subdivide the work further using 2D thread-blocks. This will under-utilize the GPU and results in increased execution times for $N_{active} < 128$.

The performance difference between CUDA and OpenCL is minimal which indicates that the compute part of both implementations inhabits similar behavior. For most values of N_{active} the timings of Sapporo1 and Sapporo2 are comparable. Only for $N_{active} < 64$ we see a slight advantage for Sapporo1 where the larger data loads of Sapporo2 result in a slightly longer execution time. However, the improvements made in Sapporo2 result in higher performance and a more responsive execution time compared to Sapporo1 when $N_{active} \geq 128$. For the HD7970, there is barely any improvement when N_{active} decreases from 256 to 128. There is a slight drop in the execution time at $N_{active} = 192$ which coincides with one less active wavefront compared to $N_{active} = 256$. When $N_{active} \leq 128$ we can launch 2D blocks and the performance improves again and approaches that of the NVIDIA hardware, but the larger wavefront size compared to the warp size causes the the execution times to be less responsive to changes of N_{active} .

2.3.3 Range of N

Now that we selected the thread-block configuration we continue with testing the performance when computing the gravitational forces using N_{sink} particles and N_{source} particles, resulting in $N_{sink} \times N_{source}$ force computations. The results are presented in the left panel of Fig. 2.3. This figure shows the results for the five GPUs using CUDA, OpenCL, Sapporo1 and Sapporo2. The execution time includes the time required to send the input data and retrieve the results from the device.

The difference between Sapporo1 and Sapporo2 (both the CUDA and OpenCL versions) on the K20m GPU are negligible. Sapporo1 is slightly faster for $N < 10^4$, because of the increased data-transfer sizes in Sapporo2, which influence the performance more when the number of computations is relatively small. Sapporo2 is slightly faster than Sapporo1 when $N \geq 10^4$, because of the various optimizations added to the new version. The difference between the GTX680, K20m and the HD7970 configurations is relatively small. While the GTX Titan is almost $1.5\times$ faster and the GTX480 almost $2\times$ slower than these three cards. These numbers are not unexpected when inspecting their theoretical per-



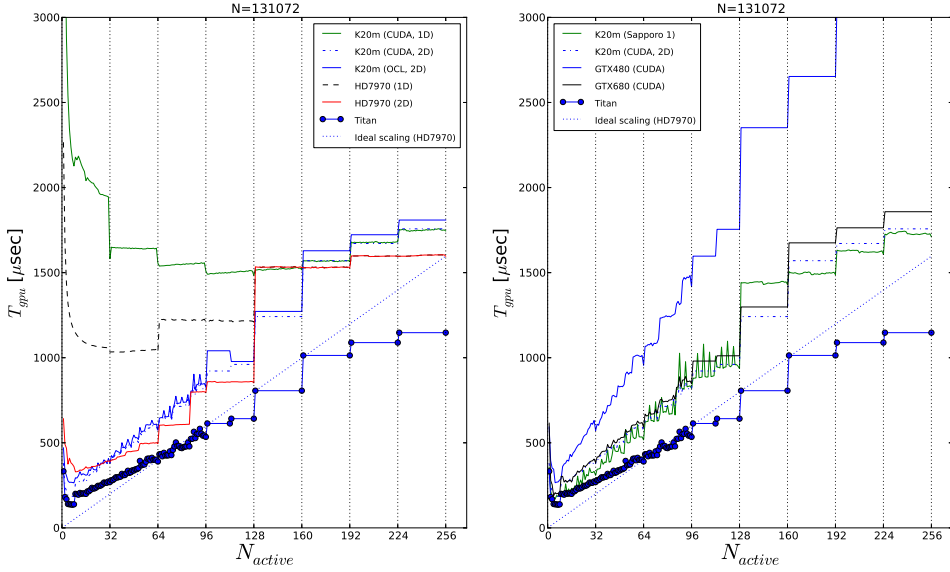


Figure 2.2: Performance for different numbers of active sink particles. The x-axis indicates the number of active particles and the y-axis the required time to compute the gravitational force using 131072 source particles ($N_{\text{active}} \times N$ gravity computations). The presented time only includes the time required to compute the gravity, the data transfer times are not included. In both panels the linear striped line shows the ideal scaling from the most optimal configuration with 256 active particles to the worst case situation with 1 active particle for one of the shown devices. The left panel shows the effect on the performance when using 1D thread-blocks instead of 2D on AMD and NVIDIA hardware. Furthermore we show the effect of using OpenCL instead of CUDA on NVIDIA hardware. When using 1D thread-blocks the GPU becomes underutilized when N_{active} becomes smaller than ~ 128 . This is visible as the execution time increases while N_{active} becomes smaller. The right panel compares the performance of the five different GPUs as indicated. Furthermore it shows that the performance of Sapporo2 is comparable to that of Sapporo1.

formance (see Tab. 2.1). For $N < 10^5$ we further see that the performance of the HD7970 is lower than for the NVIDIA cards. This difference is caused by slower data transfer rates between the host and device for the HD7970. Something similar can be seen when we compare the OpenCL version of the K20m with the CUDA version. Close inspection of the timings indicate that this difference is caused by longer CPU-GPU transfer times in the OpenCL version when transferring small amounts of data (< 100 KB), which for small N forms a larger part of the total execution time.

2.3.4 Double precision vs Double-single precision

As mentioned in Section 2.2.2 the higher order integrators require the use of double precision computations. Therefore we test the performance impact when using full native double precision instead of double-single precision. For this test we use the GTX680, K20m and the HD7970. The theoretical peak performance when using double precision computations is lower than the peak performance when using single precision computations. The double

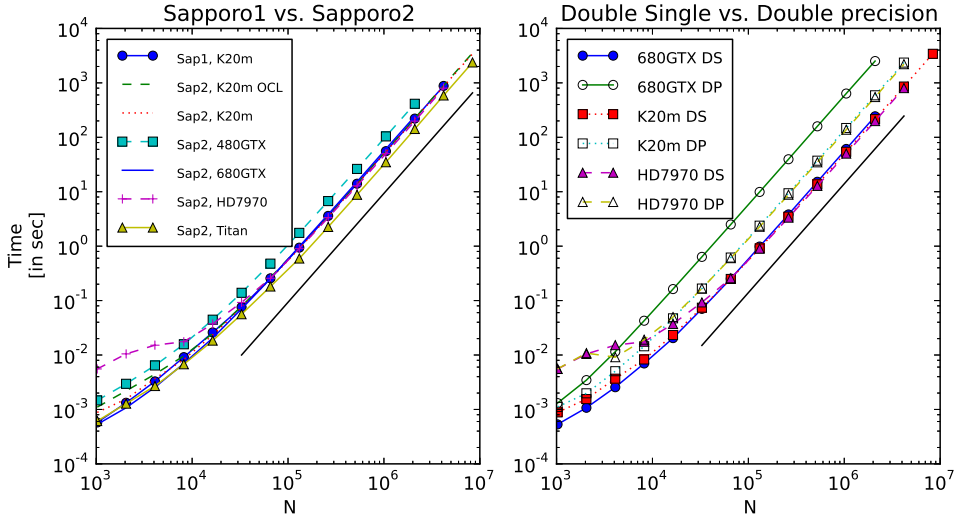


Figure 2.3: Time required to solve N^2 force computations using different configurations. In both panels the number of source particles is equal to the number of sink particles which is indicated on the x-axis. The y-axis indicates the required wall-clock time to execute the gravity computation and to perform the data transfers. Unless otherwise indicated we use CUDA for the NVIDIA devices. The left panel shows the performance of Sapporo1 on a K20m GPU and Sapporo2 on 5 different GPUs using a mixture of CUDA and OpenCL. The straight solid line indicates N^2 scaling. The right panel shows the difference in performance between double-single and double precision. We show the performance for three different devices. The double-single timings are indicated by the filled symbols. The double-precision performance numbers are indicated by the lines with the open symbols. The straight solid line indicates N^2 scaling.

precision performance of the K20m is one third that of the single precision performance. For the GTX680 this is $\frac{1}{24}$ th and for the HD7970 this is one fourth. As in the previous section we use the wall-clock time required to perform N^2 force computations to compare the devices. The results are presented in the right panel of Fig. 2.3, here the double precision timings are indicated with the open symbols and the double-single timings with the closed symbols.

As in the previous paragraph, when using double-single precision the performance is comparable for all three devices. However, when using double-precision the differences become more clear. As expected, based on the theoretical numbers, the GTX680 is slower than the other two devices. The performance of the K20m and the HD7970 are comparable for $N > 10^4$. For smaller N the performance is more influenced by the transfer rates between the host and the device than by its actual compute speed.

Taking a closer look at the differences we see that the performance of the GTX680 in full double precision is about $\sim 10\times$ lower than when using double-single precision. For the other two cards the double precision performance is roughly $\sim 1.5\times$ lower. For all the devices this is roughly a factor of 2 difference from what can be expected based on the specifications. This difference can be explained by the knowledge that the number of oper-

ations is not exactly the same for the two versions⁴ and even in the double single method we use the special operation units to compute the `sqrt`. Another reason for the discrepancy between the practical and theoretical numbers is that we keep track of the nearest neighbours which requires the same operations for the double single and the double precision implementation. Combining this with the knowledge that we already execute a number of double precision operations to perform atomic additions and data reads, results in the observed difference between the theoretical and empirically found performance numbers.

2.3.5 Sixth order performance

The reason to use sixth order integrators compared to lower order integrators is that, on average, they are able to take larger time-steps. They are also better in handling systems that contain large mass ratios (for example when the system contains a supermassive black-hole). The larger time-step results in more active particles per block-step which improves the GPU efficiency. However, to accurately compute the higher order derivatives double precision accuracy has to be used. This negatively impacts the performance and in Fig. 2.4 we show just how big this impact is. As in the previous figures we present the time to compute N^2 forces. Presented are the performance of the sixth order kernel using double precision, the fourth order kernel using double-single precision and the fourth order kernel using double precision. As expected, the sixth order requires the most time to complete as it executes the most operations. The difference between the fourth order in double-single and the sixth order in double precision is about a factor 4. However, if we compare the performance of the double precision fourth order kernel with the sixth order kernel the difference is only about a factor of 1.4. This small difference in performance shows that it is beneficial to consider using a sixth order integrator when using high mass ratios or if, for example, high accuracy is required to trace tight orbits.

2.3.6 Multi-GPU

As described in Section 2.2, Sapporo2 supports multiple GPUs in parallel. The parallelised parts are the force computation, data transfer and prediction of the source particles. The transfer of particle properties to the device and the transfer of the force computation results from the device are serial operations. These operations have a small but constant overhead, independent of the number of GPUs. For the measurements in this section we use the total wallclock time required to compute the forces on N particles (as in Section 2.3.3). The speed-up compared to 1 GPU is presented in Fig. 2.5. The timings are from the K20m GPUs which have enough memory to store up to 8×10^6 particles. For $N > 10^4$ it is efficient to use all available GPUs in the system and for $N \leq 10^4$ all multi-GPU configurations show similar performance. The only exception here is when $N = 10^3$ at which point the overhead of using 4 GPUs is larger than the gain in compute power. For large enough N the scaling is near perfect ($T_{\text{single-GPU}}/T_{\text{multi-GPU}}$), since the execution time is dominated by the computation of the gravitational interactions.

⁴Double single requires more computations than single precision on which the theoretical numbers are based

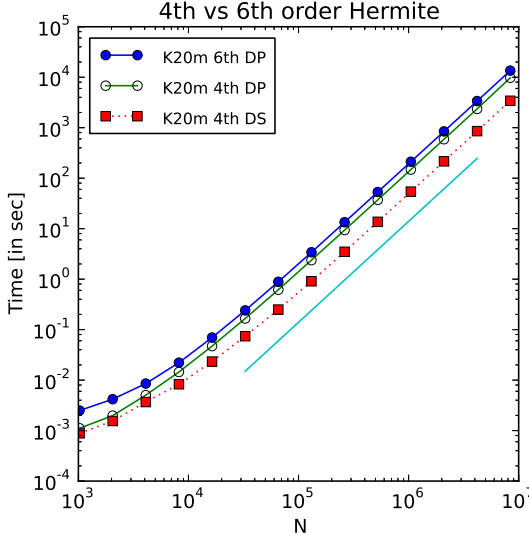


Figure 2.4: Performance difference between fourth and sixth order kernels. Shown is the time required to solve N^2 force computations using different configurations. The number of source particles is equal to the number of sink particles indicated on the x-axis. The y-axis indicates the required wall-clock time to execute the gravity computation and to perform the data transfers. The fourth-order configuration using double-single precision is indicated by the dotted line with square symbols, the fourth-order configuration using double precision is indicated by the solid line with open circles and the solid line with closed circles indicates a sixth-order configuration using double precision. The straight solid line without symbols indicates the N^2 scaling. Timings performed on a K20m GPU using CUDA 5.5.

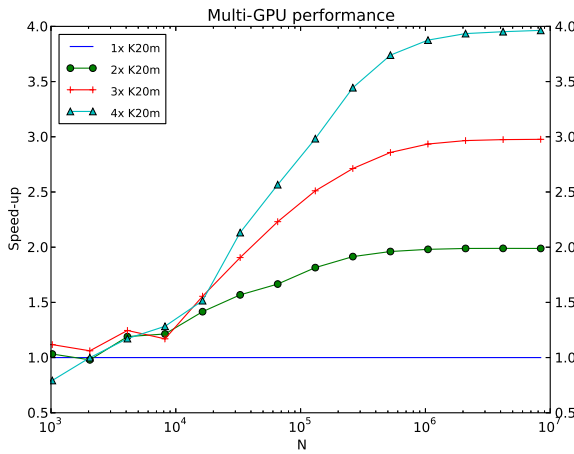
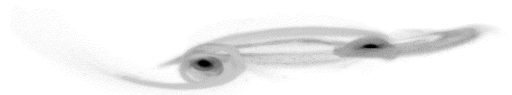


Figure 2.5: Multi-GPU speed-up over using one GPU. For each configuration the total wall-clock time is used to compute the speed-up (y-axis) for a given N (x-axis). The wall-clock time includes the time required for the reduction steps and data transfers. Timings performed on K20m GPUs using Sapporo2 and CUDA 5.5.



2.4 Discussion and CPU support

2.4.1 CPU

With the availability of CPUs with 8 and more cores that support advanced vector instructions there is the recurring question if it is not faster to compute the gravity on the CPU than on the GPU. Especially since there is no need to transfer data between the host and the device which can be relatively costly when the number of particles is ≤ 1024 . To test exactly for which number of particles the CPU is faster than the GPU we added a CPU implementation to Sapporo2. This CPU version has support for SSE2 vector instructions and OpenMP parallelisation. The only kernel implemented is the fourth order integrator, including support for neighbour lists and nearest neighbours (particle-ID and distance). Because the performance of the GPU depends on the combination of sink and source particles we test a grid of combinations for the number of sink and source particles when measuring the time to compute the gravitational forces. The results for the CPU (a Xeon E5620 @ 2.4Ghz), using a single core, are presented in Fig. 2.6a. In this figure (and all the following figures) the x-axis indicates the number of sinks and the y-axis the number of sources. The execution time is indicated by the colour from blue (fastest) to red (slowest). The smooth transition from blue to red from the bottom left corner to the top right indicates that the performance does not preferentially depend on either the source or sink particles but rather on the combined number of interactions. This matches our expectations, because the parallelisation granularity on the CPU is as small as the vector width, which is 4. On the GPU this granularity is much higher, as presented in Fig. 2.6b, here we see bands of different colour every 256 particles. Which corresponds to the number of threads used in a thread-block ($N_{threads}$). With 256 sink particles we have the most optimal performance of a block, if however we would have 257 sink particles we process the first 256 sinks using optimal settings while the 257th sink particle is processed relative inefficient. This granularity becomes less obvious when we increase the number of interactions as presented in Fig. 2.6c. Here we see the same effect appearing as with the CPU (Fig. 2.6a), where the granularity becomes less visible once we saturate the device and use completely filled thread-blocks for most of the particles. The final panel, Fig. 2.6d, indicates per combination of source and sink particles which CPU or GPU configuration is the fastest. For the CPU we measured the execution time when using 1,2,4 or 8 cores. In this panel the colours indicate the method which gives the shortest execution times.

When either the number of sinks or the number of sources is relative small (≤ 100) the CPU implementation performs best. However, when the number of sinks or sources is > 100 the GPU outperforms the CPU. When using a CPU implementation that uses the AVX or AVX2 instruction sets the borders of these regions would shift slightly upwards. The CPU would then be faster for a larger number of source/sink particles, but that would only be at most a factor of 2 to 4 more particles. The data of Fig. 2.6 confirms that our choice to implement the Sapporo2 library for the GPU is an efficient method for realistic data-set sizes.

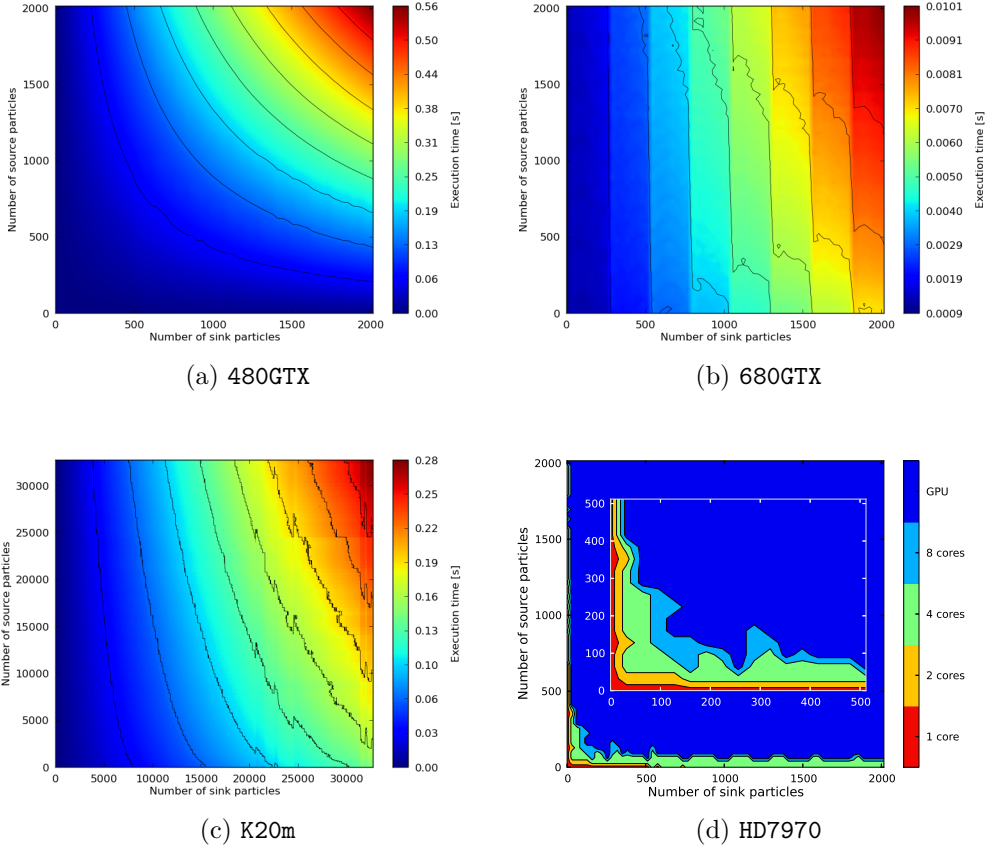
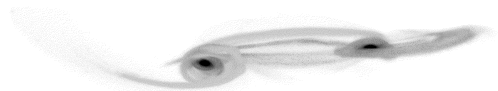


Figure 2.6: GPU and CPU execution times. In all the subplots the x-axis indicates the number of sink particles and the y-axis the number of source particles used. For subplots a,b and c the raw execution times are presented and indicated with the colours. Plot d does not present the execution time but rather which of the configuration gives the best performance. The inset of plot d is a zoom-in of the main plot. Note that the colours are scaled per plot and are not comparable between the different subplots. All the GPU times include the time required to copy data between the host and device.

2.4.2 XeonPhi

Because the Sapporo2 library can be built with OpenCL it should, theoretically, be possible to run on any device that supports OpenCL. To put this to the test, we compiled the library with the Intel OpenCL implementation. However, although the code compiled without problems it did not produce correct answers. We tested the library both on an Intel CPU and the Intel XeonPhi accelerator. Neither the CPU, nor the XeonPhi produced correct results. Furthermore, the performance of the XeonPhi was about $100\times$ lower than what



can be expected from its theoretical peak performance. We made some changes to the configuration parameters such as $N_{threads}$ and N_{blocks} , however this did not result in any presentable performance. We suspect that the Intel OpenCL implementation, especially for XeonPhi, contains a number of limitations that causes it to generate bad performing and/or incorrect code. Therefore the Sapporo2 library is not portable to Intel architectures with their current OpenCL implementation. This does not imply that the XeonPhi has bad performance in general, since it is possible to achieve good performance on N -body codes that is comparable to GPUs. However, this requires code that is specifically tuned to the XeonPhi architecture (K. Nitadori, private communication ⁵).

2.5 Conclusion

The here presented Sapporo2 library makes it easy to enable GPU acceleration for direct N -body codes. We have seen that the difference between the CUDA and OpenCL implementations is minimal when there are enough particles to make the simulation compute limited. However, if many small data transfers are required, for example when the integrator takes very small time-steps with few active particles, the CUDA implementation will be faster. Apart from the here presented fourth and sixth order integrators the library also contains a second order implementation. And because of the storage of data in double precision it can be trivially expanded with an eight order integrator. The performance gain when using multiple GPUs implies that it is efficient to configure GPU machines that contain more than 1 GPU. This will improve the time to solution for simulations with more than 10^4 particles.

The OpenCL support and built-in tuning methods would allow easy extension to other OpenCL supported devices. However, this would require a mature OpenCL library that supports atomic operations and double precision data types. For the CUDA devices this is not that a problem since the current CUDA libraries already have mature support for the used operations and the library automatically scales to future architectures. The only property that has to be set is the number of thread-blocks per multiprocessor and this can be easily identified using the figures as presented in Section 2.3.1.

Acknowledgements

This work was supported by the Netherlands Research Council NWO (grants #643.200.503, #639.073.803, #614.061.608, #612.071.503, #643.000.802).

⁵Also see <https://github.com/nitadori/Hermite> and <http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx>.

