HAL
archives-ouvertes.fr

Dual tree traversal on integrated GPUs for astrophysical N-body simulations

Pierre Fortin, Maxime Touche

▶ To cite this version:

Pierre Fortin, Maxime Touche. Dual tree traversal on integrated GPUs for astrophysical N-body simulations. International Journal of High Performance Computing Applications, SAGE Publications, 2019, 33 (5), pp.960-972. 10.1177/1094342019840806 . hal-02073710

HAL Id: hal-02073710
https://hal.sorbonne-universite.fr/hal-02073710

Submitted on 23 Jul 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Dual tree traversal on integrated GPUs for astrophysical N-body simulations

Pierre Fortin[ab] and Maxime Touche[a]

[a]Sorbonne Université, UPMC Univ Paris 06, CNRS,
LIP6 UMR 7606, F-75005 Paris, France
[b]Univ. Lille, CNRS, Centrale Lille, CRIStAL UMR 9189, F-59000 Lille, France
Email: pierre.fortin@sorbonne-universite.fr

July 23, 2020

Abstract

In astrophysical N-body simulations, O(N) fast multipole methods (FMMs) with dual tree traversal (DTT) on multi-core CPUs are faster than O(N log N) CPU tree-codes but can still be outperformed by GPU ones. In this paper, we aim at combining the best algorithm, namely FMM with DTT, with the most powerful hardware currently available, namely GPUs. In the astrophysical context requiring low accuracies and non-uniform particle distributions, we show that such combination can be achieved thanks to an hybrid CPU-GPU algorithm on integrated GPUs: while the DTT is performed on the CPU cores, the far- and near-field computations are all performed on the GPU cores. We show how to efficiently expose the interactions resulting from the DTT to the GPU cores, how to deploy both the far- and near-field computations on GPU and how to overlap the parallel DTT on CPU with GPU computations. Based on the *falcON* code and using OpenCL on AMD Accelerated Processing Units and on Intel integrated GPUs, this first heterogeneous deployment of DTT for FMM outperforms standard multi-core CPUs, and matches GPU and high-end CPU performance, being hence more cost- and power-efficient.

Keywords: dual tree traversal, integrated GPU, hybrid CPU-GPU algorithm, fast multipole method, astrophysics

1 Introduction

The N-body problem describes the computation of all pairwise interactions among N bodies (or particles). Once computed, the corresponding forces are used to update the body positions and velocities for the next time-step. In astrophysics, such N-body simulations are essential and widely used for galactic dynamics studies. The gravitational force computation is the most time-consuming part and limits in practice the number of bodies, which is currently much smaller than the number of stars in a real galaxy.

The direct computation of all pairwise interactions among N bodies leads to a prohibitive $\mathcal{O}(N^2)$ runtime complexity. Thanks to the mutuality of gravity (or symmetry of Newton's third law), which states that the force of a particle A on a particle B is the opposite of the force of B on A, one can halve the computation cost, but this latter remains too expensive for millions of particles. Hierarchical methods [Barnes and Hut, 1986, Cheng et al., 1999] have therefore been introduced to reduce this runtime complexity: thanks to an octree data structure, the force field

is decomposed in a near-field part, directly computed, and a far-field part approximated with various expansions. In astrophysics, the $O(N \lg N)$ Barnes-Hut tree-code is one of the most used parallel algorithms on CPUs [Springel, 2005], and also on GPUs [Bédorf et al., 2012, 2014, Burtscher and Pingali, 2011] with increased performance compared to CPUs.

Dehnen's fast multipole method (FMM) [Dehnen, 2002] is one order of magnitude faster than serial executions of Barnes-Hut tree-codes [Dehnen, 2002, Fortin, P. et al., 2011], thanks to the $O(N)$ FMM operation count, to the use of cartesian Taylor expansions and to its dual tree traversal (DTT). Parallel implementations of this DTT-based FMM have been successfully obtained on multi-core CPUs thanks to task programming [Dehnen, 2014, Lange and Fortin, 2014, Taura et al., 2012]. Along with SIMD processing, these can compete with GPU tree-codes but usually still lag behind [Lange and Fortin, 2014, Yokota, 2013].

In this paper, we aim at combining the best algorithm, namely FMM with DTT, with the most powerful hardware currently available, namely GPUs, for astrophysical simulations. This is achieved thanks to an hybrid CPU-GPU algorithm deployed on integrated GPUs (iGPUs), such as AMD APUs (Accelerated Processing Units) and Intel iGPUs.

1.1 Related work and positioning

Within fast multipole methods, the dual tree traversal enables a simple and flexible yet very efficient octree traversal, which naturally adapts to non-uniform distributions of particles [Teng, 1998, Warren and Salmon, 1995, Yokota, 2013]. Such DTT-based FMMs have been first applied to astrophysics with the serial *falcON* code (*Force ALgorithm with Complexity* $\mathcal{O}(N)$) of W. Dehnen [2002], but then also to molecular dynamics [Coles and Masella, 2015, Lorenzen et al., 2012].

We rely on the *pfalcON* code[1] where the DTT is performed in parallel on multi-core CPUs thanks to task synchronizations based on atomic operations [Lange and Fortin, 2014]. Another task-based DTT parallelization has been achieved thanks to a rewriting of the DTT [Taura et al., 2012] and implemented in the `exaFMM` code[2]. Both codes offer very good scaling on multi-core CPU architectures, as well as on many-core ones like Intel Xeon Phi processors [Abduljabbar et al., 2017, Lange and Fortin, 2014]. Since the *falcON* code is dedicated to astrophysical simulations, *pfalcON* is slightly faster than `exaFMM` for such simulations [Lange and Fortin, 2014].

While the FMM has already been deployed on GPUs in numerous works [Choi et al., 2014, Gumerov and Duraiswami, 2008, Hamada et al., 2009, Hu et al., 2011, Lashuk et al., 2009, Overman et al., 2013, Rahimian et al., 2010, Yokota and Barba, 2011], none of these works applies to the DTT-based FMM. Indeed due to its double recursion, obtaining an efficient DTT on many-core architectures like GPUs is difficult. And to our knowledge, no other FMM variant has been shown to be faster for astrophysical simulations than the DTT-based FMM with cartesian Taylor expansions.

In Yokota and Barba [2011] the FMM deployment on GPU is performed by concatenating all source particles and expansions in a large buffer that is then transfered over the PCI bus to the discrete GPU. This approach is well-suited for moderately or highly accurate FMMs (e.g. for molecular dynamics), where the average relative error with respect to the forces obtained by direct summation (i.e. exact computations without FMM approximation) has to be small enough. However in our astrophysical context, we can rely on a low accuracy FMM where the average relative force error can be between 10^{-2} and 10^{-3}. This implies a low number of expansion terms (6 in the multipole expansions and 20 in the local expansions), as well as a low

[1] Available at: `https://pfalcon.lip6.fr`

[2] See: `https://github.com/exafmm/exafmm`

Table 1: Data requirements and estimated transfer time (considering transfers from host to GPU and from GPU to host) over a PCI-E 3.0 (considering a sustained bandwidth of 12 GB/s) for a 10M Plummer distribution on a discrete GPU.

Source particles	2.67 GB
Target particles	0.30 GB
Multipole expansions	3.07 GB
Local expansions	0.09 GB
Transfer time	0.53 s

number of particles per cell (up to 32 on CPU). These in turns reduce the compute intensity of both far- and near-field computations. In the end, as shown in Table 1, following the strategy described in Yokota and Barba [2011] leads in our case to a too large share of PCI transfer time in the overall computation: we emphasize indeed that in order to compete with high-end CPU and GPU implementations, we have to target less than 1s for 10M particles [Lange and Fortin, 2014]. Moreover, the time required on CPU to copy all the source data will also become non negligible for such low accuracies, and we may also face memory problems on the limited GPU memory when N increases. In order to overcome these issues, we will thus aim at minimizing the data volume exchanged by the CPU and the GPU, while relying on integrated GPUs to reduce the data exchange cost. We are not aware of any previous work regarding the FMM deployment on integrated GPUs.

1.2 Contributions

In this paper, we present the first CPU-GPU heterogeneous deployment of a fast multipole method based on dual tree traversal, using integrated GPUs. This is also, to our knowledge, the first FMM deployment on integrated GPUs. Such deployment requires a new hybrid CPU-GPU algorithm, where the DTT is performed on the CPU cores, and all the computations are performed on the iGPU cores.

We propose and discuss different strategies, along with their data-structures, to expose all interactions resulting from the DTT performed on the CPU cores to the iGPU cores (Sects. 3 & 4). We also show how to efficiently perform the two main computations (the far-field and near-field ones) on the iGPU in our specific context: due to the low accuracy required in astrophysics, we have here numerous independent fine-grain computations, with largely varying particle numbers for the near-field part (Sects. 3 & 4). We also detail the impact of the architecture differences between the AMD APU and the Intel iGPU on our optimizations and on their performance. Relying on the task-based parallelism introduced in *pfalcON*, we also design a parallel DTT that can be overlapped with the GPU computations (Sect. 5).

All this is implemented in OpenCL in the *pfalcON* code (based itself on the *falcON* code) as *pfalcON-iGPU*, and a performance comparison is finally performed between *pfalcON-iGPU* on integrated GPUs, *pfalcON* on multicore CPUs and the `Bonsai` code (currently one of the fastest GPU tree-codes [Bédorf et al., 2012, 2014]) on GPU (Sect. 6).

We first start with a description of N-body algorithms, especially Dehnen's algorithm, in Sect. 2 where we also present these integrated GPUs.

2 N-body algorithms and integrated GPUs

2.1 N-body algorithms

We focus here on galactic simulations and on hierarchical N-body algorithms. We do not consider other N-body algorithms such as those based on Particle-Mesh methods [Springel, 2005] for example. In the hierarchical methods, the 3D particle space is hierarchically decomposed by means of an octree. This octree is built by inserting particles one by one and by subdividing octree leafs containing more than a given maximum number of particles, denoted by *Ncrit*. The optimal *Ncrit* is usually practically determined and roughly balances the direct (near-field) computation cost with the approximate (far-field) computation cost.

2.2 Barnes-Hut tree-codes

The Barnes-Hut tree-code algorithm [Barnes and Hut, 1986] computes the gravitational forces among N particles with a $\mathcal{O}(N \ln N)$ runtime complexity thanks to monopole (and possibly quadrupole) moments. For each target body, the octree is here recursively traversed and "body-cell" or "body-body" interactions are evaluated depending on the multipole acceptance criterion (MAC):

$$\frac{D}{r} < \theta \,,$$

where D denotes the octree cell side length, r is the distance from the target body to the cell center of mass, and θ is an input parameter that balances accuracy and computation cost. Such expansions are well-suited for the relatively low accuracies required in astrophysical N-body simulations, where a relative force error of few 10^{-3} is usually adequate. The loop on the target bodies is parallel which enables CPU parallel implementations with multi-threading and/or with MPI [Springel, 2005]. This inherent parallelism has also been efficiently exploited to develop GPU implementations that run entirely on the GPU [Bédorf et al., 2012, 2014, Burtscher and Pingali, 2011]. For example, the `Bonsai` code[3], which relies on monopole and quadrupole moments and on a specific MAC, enables speedups around 20 on GPU compared to a multi-core CPU implementation [Bédorf et al., 2012].

2.3 Fast multipole methods for astrophysics

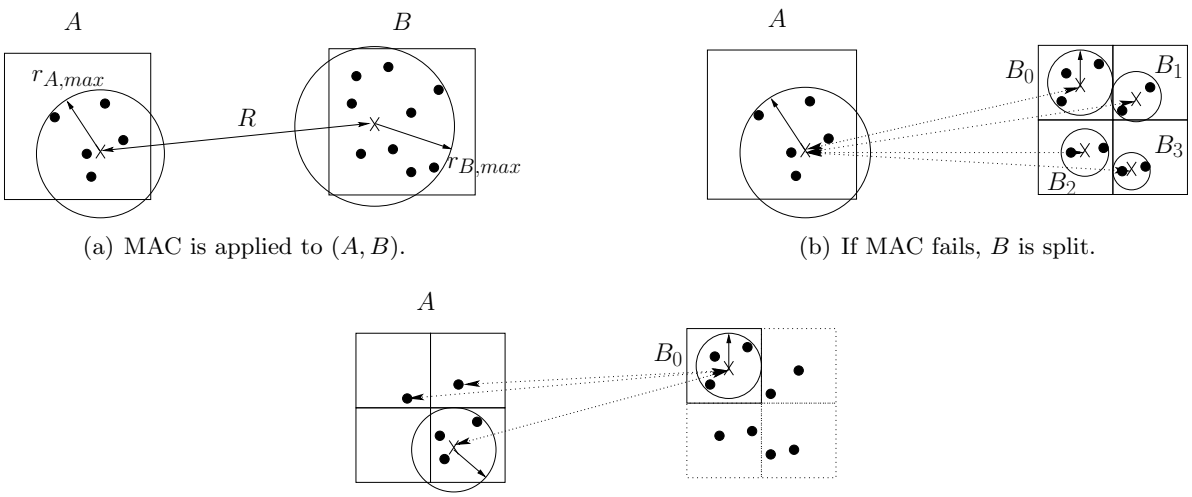
Dehnen's algorithm [Dehnen, 2002] can be considered as a fast multipole method [Cheng et al., 1999] specific to the relatively low accuracies required in astrophysics. This $\mathcal{O}(N)$ algorithm indeed relies on "cell-cell" interactions. This requires specific, low accuracy local expansions based on cartesian Taylor expansions, and a specific MAC that can balance (along with the expansion order, which is fixed to 3) the accuracy and the computation cost. This MAC is defined for two cells (A, B) (see Fig. 1(a)) as:

$$\frac{r_{A,max} + r_{B,max}}{R} < \theta \,,$$

where $r_{C,max}$ denotes an upper limit for the distance of any body within the node C from its center of mass [Dehnen, 2002]. Once the octree has been built using at most *Ncrit* particles per octree leaf, the multipole expansions are calculated during an upward pass in the octree. The interactions are then computed in the following two steps.

The first step (*interaction phase*) is presented in Algorithm 1 and relies on the dual tree traversal (DTT) presented in Fig. 1. If the MAC succeeds between two cells (A, B), their

[3]See: `https://github.com/treecode/Bonsai`



(a) MAC is applied to (A, B).

(b) If MAC fails, B is split.

(c) If MAC(A, B_0) fails, A is split.

Figure 1: Dual tree traversal in Dehnen's algorithm.

interactions can be approximated: both local expansions of A and B are updated based on the multipole expansions of A and B (*M2L* - *multipole-to-local* - operation) as shown in Fig. 1(a). More precisely, once the contribution of the multipole expansion of B on the local expansion of A has been computed, we can at low cost deduce the opposite contribution (of the multipole expansion of A on the local expansion of B) using the mutuality of *M2L* interactions. The DTT algorithm enables indeed to consider both operations at the same time. If the MAC fails, the larger cell (B here) is split and the MAC is applied between A and all the children of B (see Fig. 1(b), with 8 children in 3D). This is applied recursively, and A can then be split when the MAC fails with A as the larger cell (see Fig. 1(c)). This thus leads to a dual recursive traversal of the octree. When the MAC fails for two octree leafs, or when the number of particles is too low (depending on empirical thresholds [Dehnen, 2002]), the direct computation (*P2P* - *particle-to-particle* - operation) is used instead of the expansions. Thanks to this DTT, Dehnen's algorithm consistently uses the mutuality of the interactions to (approximately) halve the computation cost in the near-field part as well as in the far-field part. This DTT also enables to better preserve the total momentum than tree-codes [Dehnen, 2002].

Once the interaction phase has traversed all the cells, the *evaluation phase* is used to evaluate the local expansion of each cell for each body within this cell. This is performed thanks to a (simply) recursive downward pass of the octree. The interaction and evaluation steps correspond together to the most time consuming part. The octree construction has a non-negligible computation time, but does not have to be performed at every time-step. Moreover, the DTT-based interaction step represents around 95% of the total time for both the interaction and evaluation steps, which makes it crucial for the overall performance.

2.4 *falcON* and *pfalcON* codes

Dehnen's algorithm has been implemented in the *falcON* [4] code which offers $\mathcal{O}(N)$ computation times one order of magnitude smaller than serial executions of Barnes-Hut tree-codes [Dehnen, 2002, Fortin, P. et al., 2011]. Moreover, these computation times are much less sensitive to the distribution of particles: this is very important for astrophysical simulations where the particle distributions representing galaxies or groups of galaxies are highly non-uniform.

[4] Available in `http://carma.astro.umd.edu/nemo/`

Algorithm 1 Interact(cell A, cell B)

if (MAC fails between A and B) **then**
  if ($A = B$) **then**
    for all pairs $\{a, aa\}$ of children of A **do**
      Interact(a, aa)
    end for
  else if $r_{A,max} > r_{B,max}$ **then**
    for all children a of A **do**
      Interact(a, B)
    end for
  else
    for all children b of B **do**
      Interact(A, b)
    end for
  end if
else
  Approximate interaction (A, B)
end if

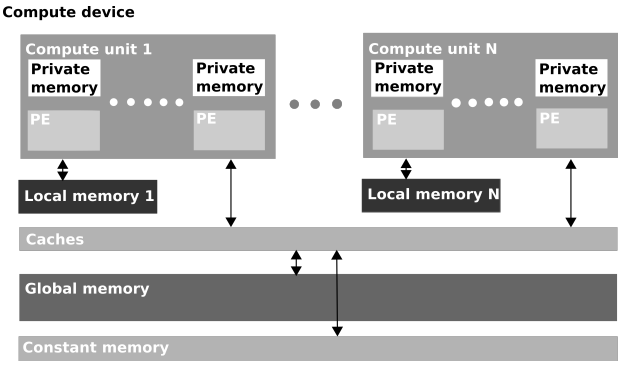


Figure 2: OpenCL compute and memory models.

In Lange and Fortin [2014], *falcON* has been parallelized in *pfalcON* on multi-core CPUs and on Intel Xeon Phi thanks to task-based parallelism (e.g. with OpenMP). The mutuality of the interactions is here fully preserved, but introduces write conflicts among tasks that are updating the local expansion of the same cell via different *M2L* operations. These conflicts are handled via atomic operations emulating fast locks. Moreover, the near-field part is vectorized using the SPMD-on-SIMD model of `ispc`[5].

In the rest of the paper, we will target a reference (non-uniform) astrophysical model, the Plummer distribution [Fortin, P. et al., 2011], with 10M particles. We compute both forces and potentials, for all particles, with single precision floating point arithmetic. We also use in each code appropriate softenings for the near-field part of the gravity [Fortin, P. et al., 2011].

2.5 Integrated GPUs

We will target two integrated GPUs with different hardware features. The first is an AMD APU (A10-7850K) which associates two 2-way SMT CPU cores with eight iGPU compute units based on the scalar GCN (Graphics Core Next) micro-architecture. On AMD GPU, each compute unit contains 64 processing elements and work-items are SIMD processed by *wave-fronts* of 64 work-items. A more detailed presentation of this AMD APU can be found in Said et al. [2018].

The second integrated GPU architecture studied here is an Intel Iris Pro Graphics P6300 (GT3e, Gen8 architecture) within an Intel Xeon E3-1285L v4 processor. The four 2-way SMT CPU cores are associated with 48 compute units on the iGPU. Each compute unit contains two SIMD FPUs that can concurrently perform four 32-bit flops each. On the software side however, the SIMD width can range from 1 to 32; SIMD-8, SIMD-16, and SIMD-32 being the most common SIMD-widths in OpenCL. More details on this Intel Processor Graphics Gen8 architecture can be found in Intel [2015].

On both architectures, the iGPU shares the same die as the CPU cores and can directly access the CPU main memory via specific *zero-copy buffers* where a large fraction of the main memory can be allocated. This enables to avoid explicit copies between the main memory and the GPU memory, to alleviate the possible performance bottleneck on discrete GPUs due to

[5] *Intel SPMD Program Compiler*, see: `https://ispc.github.io/`

the PCI bus, and to allocate more memory than within a discrete GPU. Moreover, these iGPUs (along with their CPU) are usually more compute powerful than standard CPUs, but offer lower compute power and memory bandwidth than discrete GPUs [Said et al., 2018]. The possible performance gains over discrete GPUs depend thus on the application and algorithm features (frequency and volume of PCI transfers, proportion of work deported on GPU...).

We will rely on OpenCL programming, whose compute and memory models are presented in Fig. 2, since OpenCL SDKs and drivers are available on both the AMD and Intel integrated GPUs. Written with the SPMD (Single Program, Multiple Data) programming model, the OpenCL kernels are launched by the host (CPU) on the device (here the integrated GPU) within a command queue, and executed independantly among multiple work-groups. The work-groups are distributed on the different compute units, each work-group being made of work-items that are SIMD executed on the processing elements (PEs) of the compute unit. The memory model of OpenCL allows different levels of sharing between work-items: the global memory is accessible by all work-items, the local memory is shared within a work-group and register values are private to each work-item. We refer the reader to AMD [2015a,b], Intel [2017] for more details on OpenCL programming on these architectures.

Finally, the fine-grain SVM (Shared Virtual Memory) OpenCL 2.x feature supports concurrent writes in the same OpenCL buffer from both the host and the iGPU device: this could have enable some computations to be more efficiently performed on the CPU (e.g. near-field computations with very low number of particles). However, such computations better processed on CPU are minority and we prefer to avoid possible performance overheads with fine-grain SVM (e.g. to maintain coherency between CPU and GPU caches). That is why we perform all computations on the iGPU, and we thus rely here on OpenCL 1.2 only.

3 The far-field part

We start our description of the DTT-based FMM deployment on integrated GPUs with the far-field part, which translates to numerous independent *M2L* computations. However, one has to ensure that two *M2L* computations will not concurrently update the same local expansion. Indeed, due to the use of the mutuality of the *M2L* interactions to save computations in the far-field part, a *M2L* operation between cells A and B will update the local expansions of both cells and can then conflict with a *M2L* operation between cells A and C. We present here several strategies to synchronize the *M2L* computations and avoid such conflicts, considering in this section only one thread performing a serial DTT on CPU.

3.1 Preserving mutuality

We start with three strategies that preserve the mutuality of *M2L* interactions, hence saving *M2L* computation costs. These strategies are also based on the most simple data structure, rapidly written by the CPU and read by the GPU, to store the *M2L* interactions.

The first one, referred to as *atomic float*, relies on emulating atomic additions on floating-point variables. Since there is no such atomic operation in OpenCL, we emulate them with a loop over the OpenCL 1.2 "compare-exchange" atomic operation. Each *M2L* computation is then assigned to one work-item, and each local expansion coefficient is updated (in both cells) thanks to these atomic additions on floating-point variables. Contrary to other works such as Hamada et al. [2009], Yokota and Barba [2011] where each *M2L* operation was performed by a work-group, we use here only one work-item since the expansion orders are lower due to the low accuracies required in astrophysics. This results in massive, regular and rather fine-grained parallelism which is well suited for GPU processing, contrary to many GPU-based

FMMs [Gumerov and Duraiswami, 2008, Hu et al., 2011, Lashuk et al., 2009, Overman et al., 2013, Rahimian et al., 2010] where the far-field part is not efficiently processed on GPU, and hence often computed on CPU. In order to expose to the GPU cores all the *M2L* interactions that have to be performed, we simply use here a large buffer (named the *interaction buffer*) where all pairs of cell indices involved in a *M2L* operation are consecutively written by the CPU thread during the DTT. This interaction buffer is created and used as a zero-copy buffer for fast data accesses by the iGPU. We also use zero-copy buffers for buffers storing the multipole and local expansions and the octree cells (as well as for the buffer storing the particles).

The second strategy, referred to as *atomic bit*, aims at reducing the number of atomic operations performed and is close to our multi-core CPU parallelization of the DTT [Lange and Fortin, 2014]. We use here atomic operations on a specific bit within the cell data structure to emulate locks and ensure exclusive access to the cell local expansion. When a work-item needs to update the local expansion of a given cell, it first has to set this bit to 1 while checking that the bit was not already set to 1 (by another work-item): this is performed by an atomic *or* operation. In case the bit was already set to 1, we use busy waiting (with the required memory fence) since the cell update is a very fast operation. When the update is over, the bit is reset to 0 by an atomic *and* operation: such write must be preceded by a memory fence to ensure that the write is performed after the computation.

During the DTT it is however likely that a given cell will be involved in multiple consecutive, or very close, *M2L* operations. As an example if MAC(A,B) fails but MAC(A,B_i) succeeds for the 8 B_i children of B, we will have to successively perform 8 *M2L* operations with A. When considering a work-group with multiple work-items, this implies that multiple work-items within the same work-group will try to obtain exclusive access to the same cell. Since the work-groups are (partly) processed in SIMD, this will result in strong contention on our bit locks. We have therefore proposed a third strategy, extending *atomic bit*[6] and referred to as *stridden atomic bit*, where successive *M2L* operations output by the DTT are written with a given stride in memory and then in a cyclic way to fill a sub-part of the interaction buffer. On an APU, we use for example a stride of 256 and cyclically fill an array of size 8*256 before writing further in memory.

3.2 Forsaking mutuality

We now present two more strategies that forsake the mutuality of *M2L* interactions: this will prevent us from saving *M2L* computation cost, but will enable us to avoid any synchronization costs.

When considering a *M2L* operation between cells A and B, the first strategy, referred to as *sort no-mutual*, writes in the interaction buffer the two couples (A, B) and (B, A) (where the first element is the target cell) instead of the pair $\{A, B\}$ (where both elements were target and source cells) as in the previous strategies with mutuality. We then sort the interaction buffer according to the first element of all couples, and compute for each target cell C the offset in the interaction buffer of the first couple $(C, \ldots)$ and the number of such $(C, \ldots)$ couples. We then launch the *M2L* kernel with one work-item per cell: each work-item processes all *M2L* computations (without mutuality) where its corresponding cell is involved as a target cell. No two work-items are thus writing in the same local expansion.

In order to avoid the sorting as well as the computations of the offsets and of the interaction numbers, we propose a last strategy, referred to as *no-mutual*. We use here a specific data structure to store, for each target cell C, the list of source cell indices involved in a *M2L* interaction with C. This data structure is depicted in Fig. 3. The large interaction buffer

[6] Such extension could also have been applied to *atomic float*.

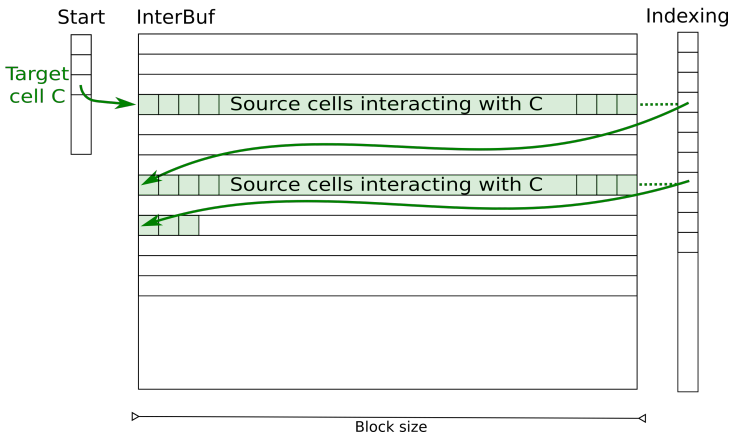


Figure 3: Interaction buffer set for *no-mutual* and *no-mutual WG* strategies.

(`InterBuf` array in Fig. 3) is here divided in blocks of several source cell indices all related to the same target cell; the block size being chosen as a cache line multiple. Thanks to two auxiliary arrays (`Start` and `Indexing` arrays in Fig. 3), storing respectively the first block index for each cell and the next block index for each block, these blocks form a linked list for each target cell C. During the DTT, when encountering a *M2L* interaction with C as the target cell, we add the source cell index to the current block in the linked list associated to C. If the block is full, we reserve a new block in the large interaction buffer and add it to the linked list. Only three arrays, referred to as *interaction buffer set*, are hence required and stored as zero-copy buffers. Our implementation has been designed so that the filling of these interaction buffers is a fast operation for the CPU. On the GPU, the work-item in charge of the target cell C has then to browse the corresponding linked list to retrieve all the source cell indices involved in a *M2L* interaction with C and to perform all these computations.

3.3 Forsaking mutuality without divergence

As the number of *M2L* interactions varies from a target cell to another, this introduces compute divergence among the work-items, hence a possible performance drop due the SIMD processing of these work-items.

We thus propose a variant for each of the two last strategies (*sort no-mutual* and *no-mutual*). Instead of having one work-item per target cell, we now use one work-group per target cell C: each work-item within this work-group will process one *M2L* computation with C as target cell. The two corresponding strategies are named *sort no-mutual WG* and *no-mutual WG*. It can be noticed that for *no-mutual WG*, the block size in the large interaction buffer has to be a work-group size multiple to ease the GPU browsing of the linked list.

One has of course to perform a reduction, thanks to the OpenCL local memory, on the local expansion terms among all work-items of each work-group. Here distinct implementations are required for the APU and the Intel iGPU due to architectural differences.

Reductions on AMD APU. Since there are 64 work-items per wave-front on the AMD APU, and 20 terms in the local expansions used in *falcON*, our best reduction implementation on APU is specific to this configuration. Moreover, the GPU local memory of the APU contains 32 banks, each bank being four bytes wide (which corresponds to a local expansion term),

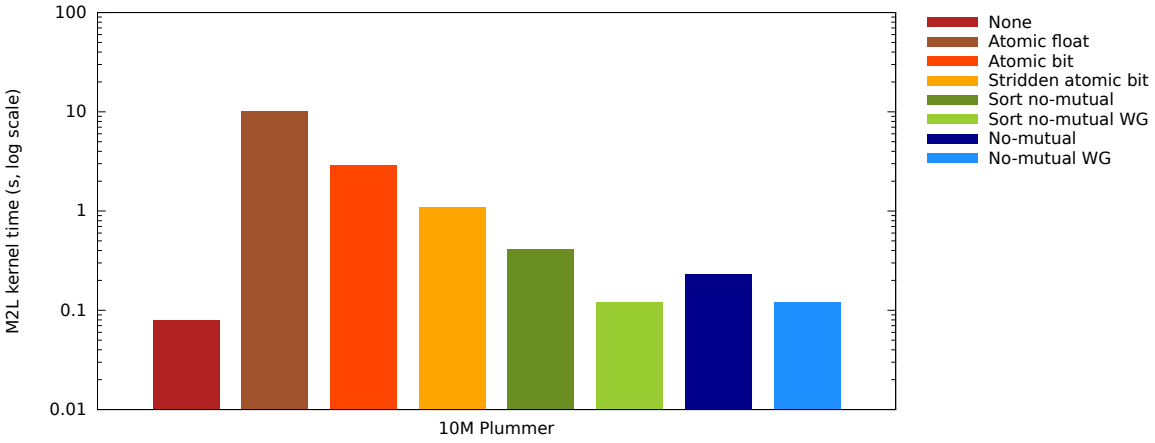


Figure 4: Performance comparison of the *M2L* synchronizations on the APU. `None` indicates a performance lower bound with a *M2L* kernel using mutuality of the interactions but no synchronizations (hence with wrong results).

and bank conflicts being determined by the memory addresses accessed by each half wavefront of 32 work-items [AMD, 2015a]. Considering a 1D work-group size of 64, we thus split the wave-front in two subparts of 32 work-items, and within each subpart each of the 20 first work-items accumulates the contributions of its subpart for a given local expansion term. The loop is manually unrolled with a factor of 4. A final step enables the 20 first work-items of the work-group to accumulate the contributions of the two subparts in the local expansion stored in global memory. Hence, no bank conflict are issued and we keep 62.5% of the work-items busy for most of the reduction.

In order to assess the efficiency of our reductions we compare the performance of the *M2L* kernel with a theoretical lower bound performance where reductions are performed in local memory without synchronizations (hence leading to wrong results): our specific reduction leads to an overhead of 28%, which is satisfactory since the *M2L* kernel time will be minority in the overall iGPU computation (see Sect. 5.1).

Reductions on Intel iGPU. We implement also specific reductions on Intel iGPU, here for all 1D work-group sizes equal to power of 2. On such architecture, the local memory is organized as 16 banks, each bank being four bytes wide (which corresponds to a local expansion term). Let s be the 1D work-group size, and b be the minimum between s and 16, the 16 first local expansions terms are processed by blocks of b terms. Each term in this block is reduced by work-group subparts of at most 16 work-items in a similar way as for our specific APU reductions. The remaining 4 last terms are also processed in a similar way using subparts of 4 work-items, each work-item partly accumulating a given local expansion term. Loops are also manually unrolled with a factor of 4.

Compared to the theoretical lower bound performance of (wrong) reductions without synchronizations in local memory, we have on Intel iGPU a low overhead of 8.6%, which validates the efficiency of our reductions on Intel iGPU.

3.4 Performance results

Figure 4 presents a performance comparison of the different *M2L* synchronizations. First, the *atomic float* synchronizations are too expensive. Then, contrary to the multi-core CPU case

(see [Lange and Fortin, 2014]) where the *atomic bit* synchronizations were fast, infrequent and interleaved with other computations, this strategy implies a too strong overhead on GPU. Using *stridden atomic bit*, we can however strongly reduce the bit lock contention. But forsaking the mutuality of the interactions with the *no-mutual* strategies leads to better results. Best results are obtained with *sort no-mutual WG* and *no-mutual WG*, where the reduction overhead is largely offset by the performance gain due to the decrease in compute divergence. In the end, we prefer *no-mutual WG* over *sort no-mutual WG* (on both the APU and the Intel iGPU) since the latter has extra costs (not shown here) for sorting and for computing the offsets and the interaction numbers.

With respect to an unreachable performance lower bound exploiting the mutuality of interactions with no synchronization overhead (see the `None` bar), one can see that the *no-mutual WG* strategy is less than twice longer which confirms the relevance of this strategy.

4 The near-field part

The direct computation of the near-field part is a classical HPC kernel, and its efficient GPU deployment has been extensively studied (e.g. [Nyland et al., 2007]). Numerous implementations are publicly available: in the CUDA SDK code samples[7] (also available in OpenCL in CUDA SDK 4.1), in the AMD OpenCL SDK 2.8 and in other OpenCL tutorials[8]. They use local memory to reduce global memory accesses to the source bodies, as well as loop unrolling. It has to be noticed that all these HPC implementations do not exploit the mutuality of gravity: it is indeed more efficient to perform twice the direct computations than to introduce divergence with the mutuality of gravity. The compute core of our *P2P* kernel relies on such implementations, however these are designed for one (very) large *P2P* operation involving thousands of bodies. In our case, we rather have to deal with numerous independant *P2P* operations involving few bodies (up to 64 in practice). This relates to GPU deployments of adaptive FMMs (without DTT) such as [Hamada et al., 2009, Lashuk et al., 2009, Overman et al., 2013, Rahimian et al., 2010, Yokota and Barba, 2011].

We choose to consider one work-group per *P2P* operation, even with our low number of particles per cell. Using one work-item per *P2P* operation as in the *M2L* strategies, would have introduced a too coarse computation grain per work-item, which is not best adapted to GPU computing. This would also have prevented the use of local memory and would have introduced compute divergence among the work-items (each *P2P* operation involving different numbers of source and target bodies).

4.1 *P2P* synchronization strategies

Following the results obtained for the far-field part, we have implemented only two strategies for the near-field part. The first one (*atomic float*) uses a large interaction buffer where all pairs of cell indices involved in a *P2P* operation are consecutively (and quickly) written by the CPU thread during the DTT. We then launch one work-group per $\{A, B\}$ pair, and this work-group has to compute the direct interactions between A and B (first $A \leftarrow B$ then $A \rightarrow B$). Here emulated atomic additions on floating-point variables (see Sect. 3.1) are used to handle the write conflicts arising when two work-groups update the same cell.

[7] `http://docs.nvidia.com/cuda/cuda-samples`

[8] See for example:
`http://www.browndeertechnology.com/docs/BDT_OpenCL_Tutorial_NBody-rev3.html`, or:
`https://developer.apple.com/library/content/samplecode/OpenCL_NBody_Simulation/Introduction/Intro.html`

The second one, named *no-mutual WG*, is directly based on the *M2L no-mutual WG* strategy. The large interaction buffer stores one linked list for each target cell C, containing indices of all source cells involved in a *P2P* interaction with C (see Fig. 3). We then launch one work-group per target cell, and no synchronization is here required among work-groups. This can be related to the sparse U-List of the kernel-independent adaptive FMM (without DTT) mentioned in Lashuk et al. [2009].

4.2 *P2P* specific optimizations

For best performance, our *P2P* kernel must be able to support various work-group sizes and various *Ncrit* values. Since moreover most cells do not have *Ncrit* particles, one of our main challenges was to minimize the number of idle work-items.

Let N_{tgt} (respectively N_{src}) be the number of bodies in the target (resp. source) cell, and s be the *P2P* (1D) work-group size. When $s \geq Ncrit$, we generalize the technique presented in Nyland et al. [2007] where multiple work-items contribute to the computation of a given target body. We refer to this technique as *multi-work-item*. In our code, this work-item number is computed dynamically as $\lfloor s/N_{tgt} \rfloor$ to best adapt this technique to each *P2P* computation. When $s < Ncrit$, *multi-work-item* is not used since this leads to worse performance: the extra number of required registers implies indeed a lower GPU occupancy.

Since N_{src} can be low, and in order to best benefit from the loop unrolling, we prefer to fill the local memory array (containing s bodies) with bodies from different cells (when considering *no-mutual WG*). In Yokota and Barba [2011], this is naturally obtained since the source bodies of all source cells are stored contiguously in memory. In our case however, we only expose the source cell indices to the GPU. It is up to the *P2P* GPU kernel to dynamically concatenate in local memory the source bodies from multiple source cells scattered in global memory. In this purpose, each work-item loads in local memory the first particle index and the number of particles of one source cell. Then each work-item browses these data in order to compute the source particle index that it will have to load in local memory. Once the work-group has loaded s bodies in local memory, or when there is no more source body, we proceed with the computation, with a manual loop unrolling by a factor of 4 (best value according to our tests).

Since source bodies are processed by multiple chunks of s bodies, we also have to loop over all target bodies when $s < Ncrit$ for each chunk of source bodies. The target bodies are then also stored in local memory in order to avoid multiple global memory accesses. The $s < Ncrit$ case also complicates the own computation.

We also tried to better exploit *multi-work-item* by forcing its usage to further reduce the number of idle work-items (for example when $s/2 < N_{tgt} < s$), but to no avail.

As final remarks, the *falcON* code does not build cells with one single particle and relies directly on the body data structure (referred to as `leaf` in *falcON*). This leads us to a separate kernel launch (still using the *P2P* kernel code) for these target cells stored as `leafs`. We refer to this *P2P* kernel launch as *P2PLeafTgt*. In practice this applies to few cells and the corresponding time is negligible (see Sect. 5.1). Besides, we recall that the *falcON* code also enables *M2P* (*multipole-to-particle*) and *P2L* (*particle-to-local*) operations, especially between a cell and a single particle [Dehnen, 2002]. Since this applies to a low number of interactions, we have replaced here these *M2P* and *P2L* operations by *P2P* ones in our iGPU deployment.

4.3 Performance results

Figure 5 presents a performance comparison of the *P2P* synchronizations, using the same OpenCL code for the compute part of the kernel. The *no-mutual WG* strategy offers a shorter time than the *atomic float* strategy, and also than the `None` one based on *atomic float* but

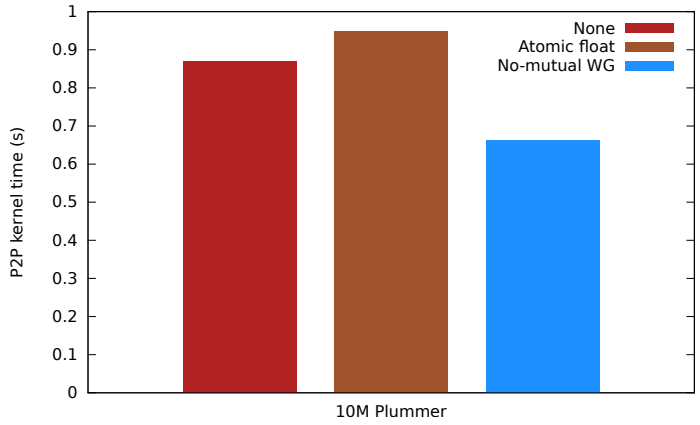


Figure 5: Performance comparison of the *P2P* synchronizations on the APU. `None` indicates a performance lower bound with a *P2P* kernel similar to *atomic float* but without atomic operations (hence leading to wrong results).

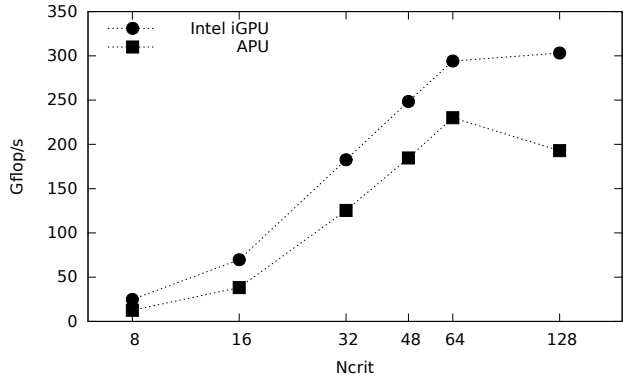


Figure 6: Performance results of the *P2P* kernel depending on *Ncrit* for a 10M Plummer distribution.

without the atomic operation cost (hence with wrong results): this is due to the fact the target cells are loaded only once from global memory with *no-mutual WG*. We thus rely hereafter on this latter for both the APU and the Intel iGPU.

We then fully optimize the *P2P* kernel as detailed in Sect. 4.2 and present our performance results in Fig. 6. Despite our *P2P* optimizations, especially for *Ncrit* lower than the work-group size, the *P2P* kernel does not perform best for cells with low body number. The *P2P* compute intensity scales indeed as $O(N)$ and the *P2P* kernel is thus memory-bound for low *Ncrit* values.

Moreover the Intel GPU outperforms the APU, which can be explained by the different SIMD widths. The 64 wide APU wave-front is a too high value for our simulations: even with $Ncrit = 64$ or $Ncrit = 128$, most cells have a particle number around 20 or 25. A non-negligible share of the work-items are then idle, even with our *multi-work-item* optimization. On the contrary, the lower and flexible SIMD width of the Intel iGPU enables (along with the work-group size) to better adapt to such cells.

In the end, the Intel iGPU reaches 303.1 Gflop/s for our numerous small *P2P* operations with very different body numbers, which represents 34% of the single precision peak performance of this iGPU. This is largely satisfactory when comparing to the CUDA N-body SDK which

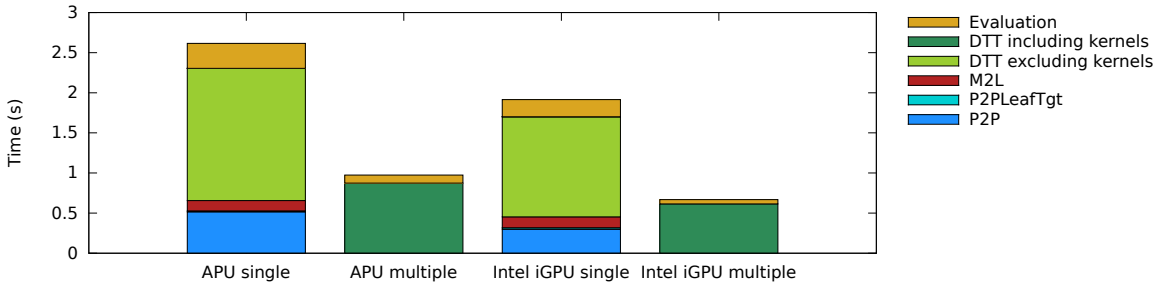


Figure 7: Performance gain due to the DTT overlap with GPU computations on a 10M Plummer distribution. `single` denotes executions with a serial DTT (using one single CPU thread) and with one single kernel launch for *P2P*, *P2PLeafTgt* and *M2L*. `multiple` denotes executions with a parallel DTT using the best number of CPU threads and the best interaction buffer size (resulting in multiple *P2P*, *P2PLeafTgt* and *M2L* kernel launchs).

reaches 40% of a NVIDIA K40c GPU peak performance for one single very large (and regular) *P2P* computation.

5 Overlapping the parallel traversal

In order to efficiently overlap the DTT on CPU with GPU computations, we rely on our task-based parallel DTT with OpenMP [Lange and Fortin, 2014]. Since no computations are performed on the CPU, we do not require here task synchronizations. Instead of filling one single interaction buffer set with multiple threads in parallel (requiring thread synchronizations), we prefer to assign one interaction buffer set to each thread: hence each thread processes its part of the DTT independently. Once one of its interaction buffer is filled, the corresponding kernel (*P2P*, *P2PLeafTgt* or *M2L*) is launched on the iGPU and the thread waits for its termination thanks to OpenCL events. We emphasize that the main goal here is to keep the iGPU device busy as much as possible, and to speed up the DTT enough so that it can be overlapped by the GPU computations. The smaller the interaction buffers are, the sooner the GPU computations will start, but the GPU computations must still be large enough to exploit all compute units and to offset the OpenCL kernel launch overhead.

Concerning the OpenCL command queues, we cannot allow two kernels of the same type (*P2P*, *P2PLeafTgt*, *M2L*) to run concurrently since this would result in write conflicts for two kernels updating the same target cell. We however can enable kernels of different types to run concurrently. The APU iGPU supports indeed concurrent execution of OpenCL kernels, but this is not the case for current Intel iGPU. Moreover, the scheduling order of multiple kernels launched concurrently by different threads is free. All this leads us to use three in-order command queues, one for each kernel type.

Finally, it can be noticed that AMD APUs offer improved GPU memory bandwidth when using read-only zero-copy buffers [AMD, 2015a]. These however are stored in USWC (Uncacheable, Speculative Write Combine) memory which implies slower CPU accesses. We tried such feature on relevant buffers but, due to intensive usage of these buffers by the CPU, this resulted overall in increased computation times on the APU.

Table 2: Architectures considered. PE stands for (OpenCL) Processing Element.

Name	Detailed name	Compute features	Launch date	TDP	Current price
APU	AMD A10-7850K Radeon R7 APU	2 2-way SMT CPU cores + 512 iGPU PE	Q1'14	95W	$150
Intel iGPU	Intel Xeon E3-1285L v4 - Iris Pro Gr. P6300	4 2-way SMT CPU cores + 384 iGPU PE	Q2'15	65W	$445
[2x]8C CPU	[2x] Intel Xeon E5-2630 v3	[2x] 8 2-wax SMT CPU cores - AVX2	Q3'14	[2x]85W	[2x]$667
18C CPU	Intel Xeon E5-2695 v4	18 2-way SMT CPU cores - AVX2	Q1'16	120W	$2424
K40c	NVIDIA K40c GPU	2880 GPU PE	Q4'13	235W	$2400

5.1 Performance results

We first performed extensive manual tuning of the following parameters: *Ncrit*, *M2L* work-group size (1D work-group) and *P2P* work-group size (1D work-group, used both in *P2P* and *P2PLeafTgt* kernels). These were performed with one single CPU thread and a large enough interaction buffer (resulting in one single kernel launch of each type). We were aiming at minimizing the sum of the *M2L*, *P2P* and *P2PLeafTgt* kernel times, without obtaining a too long DTT time on CPU (in order to still be able to overlap it). This tuning led on APU to an optimal *Ncrit* value of 48, a *M2L* work-group size of 64 and a *P2P* work-group size of 64. On Intel iGPU, the optimal *Ncrit* value is 32, with a *M2L* work-group size of 16 and a *P2P* work-group size of 48.

Based on these values, we have then determined the best number of CPU threads and the best size for our interaction buffer. The optimal values for these parameters depend on multiple factors such as the number of physical CPU cores, the number of hardware threads per CPU core, and finally the need to keep as much as possible the GPU filled with computations. In the end, best results were obtained with 6 to 8 CPU threads on the APU, with 8 CPU threads on the Intel iGPU, and with an interaction buffer size of 10^7 integer elements on both architectures.

Using these values we present in Fig. 7 the gain obtained in overlapping the DTT with GPU computations. Based on executions with one CPU thread and one kernel launch for *P2P*, *P2PLeafTgt* and *M2L*, one can see that the *P2P* kernel is here the most time consuming among the three kernels, and that the *P2PLeafTgt* time is negligible. Using a parallel DTT with multiple CPU threads and multiple kernel launches, we manage to largely overlap the DTT on CPU with GPU computations. We hence obtain final DTT times (including the kernel execution times) of 0.87s on the APU and 0.62s on the Intel iGPU. The remaining non-overlap part of the DTT represents 25% of these times (0.22s on the APU and 0.16s on the Intel iGPU).

Besides, the evaluation step is performed on the CPU cores like in Lange and Fortin [2014]: using task parallelism with multiple CPU threads, this step is a minor part in the overall time as shown in Fig. 7.

In the end, the Intel iGPU performs here better than the APU. This is due to three reasons. Firstly, the Intel iGPU compute power is 20% higher than the APU iGPU one (883.2 Gflop/s against 737.3 Gflop/s). Secondly, as shown in Sect. 4.3 the *P2P* kernel performs better on the Intel iGPU, the optimal *Ncrit* values (32 or 48) being best processed by the lower SIMD widths of the Intel iGPU than by the 64 APU one. Thirdly, the compute power of the CPU part associated with the Intel iGPU being greater than the APU one (especially thanks to twice more cores): this eases minimizing the time of the non-overlap part of the DTT, as well as the evaluation step time.

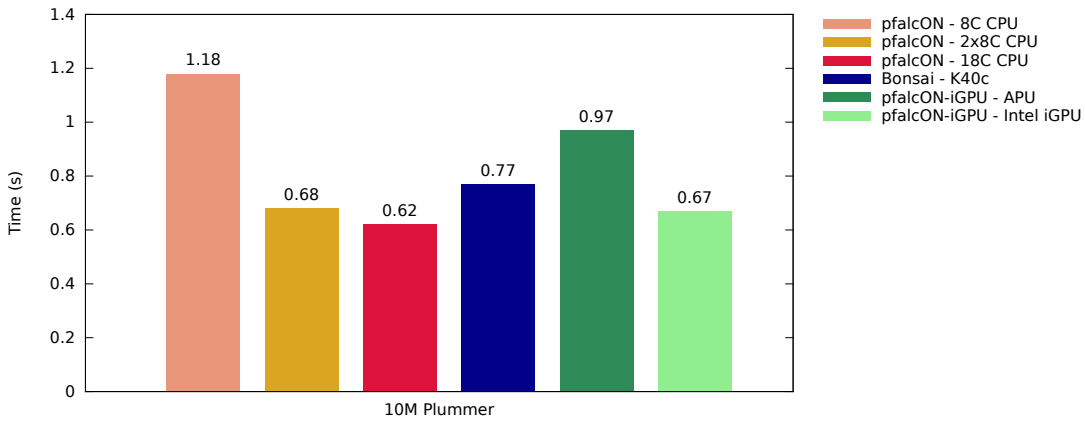


Figure 8: Computation times (interaction and evaluation steps) for *pfalcON*, *pfalcON-iGPU* and `Bonsai` on various architectures.

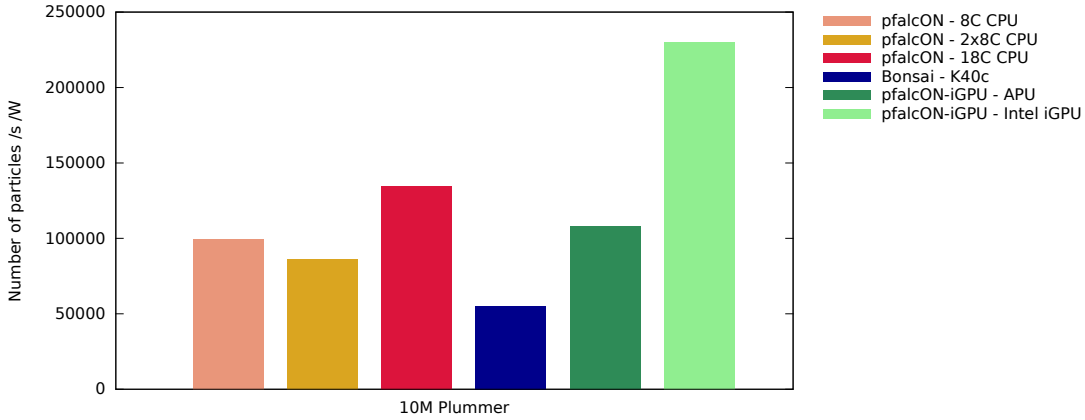


Figure 9: Power efficiency (based on the theoretical TDP values of Table 2, and on the computation times of Fig. 8) for *pfalcON*, *pfalcON-iGPU* and `Bonsai` on various architectures.

6 Comparison with CPUs and discrete GPUs

Finally we now compare *pfalcON*, our task-based multi-core CPU version of *falcON* [Lange and Fortin, 2014][9], with *pfalcON-iGPU*, which denotes our iGPU deployment of *pfalcON*, as well as with the `Bonsai` GPU code. Table 2 presents the architectures considered in this comparison. For *pfalcON*, we are using GCC+OpenMP with all available hardware threads, and `ispc` with AVX2 for *P2P* SIMD processing. `Bonsai` is built with CUDA. Optimal *Ncrit* values are used for *pfalcON* and *pfalcON-iGPU*, whereas `Bonsai` uses its own specific thresholds ($N_{leaf} = 16$ and $N_{crit} = 64$, see Bédorf et al. [2012]). As recommended for astrophysical N-body simulations [Fortin, P. et al., 2011], we use $\theta = 0.6$ for *pfalcON* and *pfalcON-iGPU*, and $\theta = 0.75$ (default value) for `Bonsai` whose expansions and MAC are different [Bédorf et al., 2012]. For *pfalcON* and *pfalcON-iGPU*, we consider the interaction and evaluation steps, and for `Bonsai` the corresponding "tree-traverse" step described in Bédorf et al. [2012] (*GPUgrav* time in the code).

First, one can notice in Fig. 8 that two (standard) 8-core CPUs with *pfalcON* outperform

[9]Since 2014, we added the new `seq_cst` clause to our OpenMP atomic operations to ensure memory barriers for our task synchronizations. This did not affected our performance results.

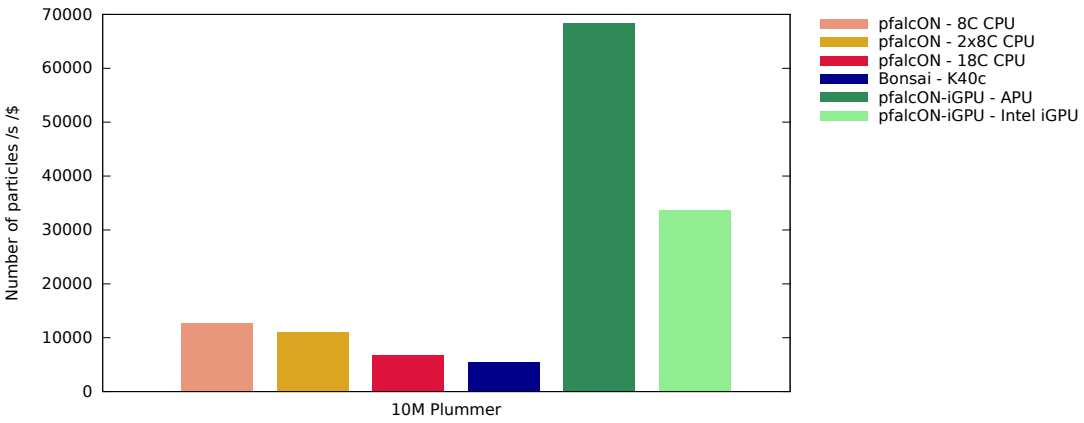


Figure 10: Cost efficiency (based on Table 2 and on Fig. 8) for *pfalcON*, *pfalcON-iGPU* and `Bonsai` on various architectures.

here `Bonsai`. More generally this depends on the exact CPU and GPU models, and better GPU performance should be obtained on newer Pascal GPUs, but we here show that CPU DTT-based FMM can match GPU tree-code performance for astrophysical simulations. This also applies to one single (high-end) 18-core CPU.

As far as *pfalcON-iGPU* and *pfalcON* are concerned, both the AMD APU and the Intel iGPU outperform a standard 8-core CPU. The Intel iGPU performance matches even the two 8-core CPU one or the 18-core CPU one (within a 7.5% margin). With respect to `Bonsai` on a K40c GPU, *pfalcON-iGPU* on the Intel iGPU is also 13% faster. We also emphasize here that 50M distributions can be run with *pfalcON* and *pfalcON-iGPU*, but not with `Bonsai`.

Morover, when considering the power efficiencies in Fig. 9, the Intel iGPU is 1.7x to 2.7x more power-efficient than the CPUs (based on the theoretical TDP values), and without considering the CPU associated with the GPU, 4.2x more power-efficient than the K40c.

When considering the cost efficiencies in Fig. 10, the Intel iGPU is 3.0x to 5.0x more cost-efficient than the CPUs, and still without considering the CPU associated with the GPU, 6.2x more cost-efficient than the K40c. Due to its very low price, the AMD APU offers here the best ratio, being 2.0x more cost-efficient than the Intel iGPU.

7 Conclusion and future work

In this paper, we have presented an hybrid CPU-GPU algorithm to deploy a fast multipole method (FMM) based on a dual tree traversal (DTT) on integrated GPUs (iGPUs) in an astrophysical context. In order to obtain the best performance results, we had to forsake the use of the mutuality of the far-field and near-field interactions in this heterogenous deployment. However, this deployment offers efficient SIMD processing of both the far- and near-field computations by aggregating multiple computations on the GPU. To our knowledge, this is the first DTT-based FMM where far-field computations are SIMD processed.

Thanks to its lower SIMD width and its greater compute-power, the Intel iGPU performs here better than the AMD APU. The Intel iGPU can match the performance of two standard CPUs, of one high-end CPU, or even of the `Bonsai` GPU tree-code, being hence up to 4.2x more power-efficient (based on the theoretical TDP values) and 6.2x more cost-efficient than these architectures.

The current work focus on one single compute node, but it would be straightforward to

extend it to multiple compute nodes, using the LET (Local Essential Tree) technique as e.g. in `exaFMM`, since all data are stored in the main memory: this is another asset compared to GPU tree-codes.

In the future, we plan to test *pfalcON-iGPU* on new integrated GPUs such as the forthcoming AMD Ryzen APUs. We also believe that our hybrid CPU-GPU algorithm could be efficiently deployed on other architectures such as integrated FPGA (with OpenCL programming), or even discrete GPUs, especially those equipped with the NVIDIA NVLink interconnect. Finally, this work could be extended to other application domains of the FMM where low accuracies are required, e.g. when using FMM as a preconditioner [Ibeid et al., 2017].

Acknowledgment

The authors would like to thank Philippe Thierry (Intel) and AMD for providing respectively the Intel iGPU and the AMD APU, Pierre-Emmanuel Le Roux (LIP6) for managing the integrated GPU nodes, as well as the master in computer science at Sorbonne Université and Charles Gueunet (LIP6) for providing access to the 18-core and 2x8-core CPUs.

References

Mustafa Abduljabbar, Mohammed Al Farhan, Rio Yokota, and David Keyes. Performance Evaluation of Computation and Communication Kernels of the Fast Multipole Method on Intel Manycore Architecture. In *Euro-Par 2017: Parallel Processing*, pages 553–564. Springer International Publishing, 2017.

AMD. *AMD APP SDK OpenCL Optimization Guide*, August 2015a.

AMD. *AMD APP SDK OpenCL User Guide*, August 2015b.

J. E. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324 (4):446–449, 1986.

Jeroen Bédorf, Evghenii Gaburov, and Simon P. Zwart. A sparse octree gravitational N-body code that runs entirely on the GPU processor. *J. Comp. Phys.*, 231(7):2825–2839, 2012.

Jeroen Bédorf, Evghenii Gaburov, Michiko S. Fujii, Keigo Nitadori, Tomoaki Ishiyama, and Simon Portegies Zwart. 24.77 Pflops on a Gravitational Tree-code to Simulate the Milky Way Galaxy with 18600 GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 54–65, 2014.

Martin Burtscher and Keshav Pingali. An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm. *GPU computing Gems Emerald edition*, pages 75–92, 2011.

H. Cheng, L. Greengard, and V. Rokhlin. A Fast Adaptive Multipole Algorithm in Three Dimensions. *Journal of Computational Physics*, 155:468–498, 1999.

Jee Choi, Aparna Chandramowlishwaran, Kamesh Madduri, and Richard Vuduc. A CPU-GPU Hybrid Implementation and Model-Driven Scheduling of the Fast Multipole Method. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 64:64–64:71, 2014.

Jonathan P. Coles and Michel Masella. The fast multipole method and point dipole moment polarizable force fields. *The Journal of Chemical Physics*, 142(2):024109, 2015.

W. Dehnen. A Hierarchical O(N) Force Calculation Algorithm. *J. Comp. Phys.*, 179:27–42, 2002.

Walter Dehnen. A fast multipole method for stellar dynamics. *Computational Astrophysics and Cosmology*, 1(1), 2014.

Fortin, P., Athanassoula, E., and Lambert, J.-C. Comparisons of different codes for galactic N-body simulations. *Astronomy & Astrophysics*, 531:A120, 2011.

N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *Journal of Computational Physics*, 227:8290–8313, 2008.

T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC'09, pages 62:1–62:12, 2009.

Qi Hu, Nail A. Gumerov, and Ramani Duraiswami. Scalable fast multipole methods on distributed heterogeneous architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 36:1–36:12, 2011.

Huda Ibeid, Rio Yokota, Jennifer Pestana, and David Keyes. Fast multipole preconditioners for sparse matrices arising from elliptic equations. *Computing and Visualization in Science*, Nov 2017.

Intel. *The Compute Architecture of Intel Processor Graphics Gen8*, 2015. Version 1.1.

Intel. *Developer Guide for Intel SDK for OpenCL Applications*, August 2017. https://software.intel.com/en-us/code-builder-user-manual.

Benoit Lange and Pierre Fortin. Parallel dual tree traversal on multi-core and many-core architectures for astrophysical N-body simulations. In *Euro-Par 2014 Parallel Processing*, volume 8632, pages 716–727. Springer International Publishing, 2014.

I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 58:1–58:12, 2009.

Konstantin Lorenzen, Magnus Schwörer, Philipp Tröster, Simon Mates, and Paul Tavan. Optimizing the accuracy and efficiency of fast hierarchical multipole expansions for md simulations. *Journal of Chemical Theory and Computation*, 8(10):3628–3636, 2012.

Lars Nyland, Mark Harris, and Jan Prins. Fast N-Body Simulation with CUDA. *GPU gems*, 3:677–695, 2007.

R. E. Overman, J. F. Prins, L. A. Miller, and M. L. Minion. Dynamic Load Balancing of the Adaptive Fast Multipole Method in Heterogeneous Systems. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1126–1135, 2013.

A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct

numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010.

Issam Said, Pierre Fortin, Jean–Luc Lamotte, and Henri Calandra. Leveraging the accelerated processing units for seismic imaging: A performance and power efficiency comparison against CPUs and GPUs. *The International Journal of High Performance Computing Applications*, 32(6):819–837, 2018.

V. Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

K. Taura, J. Nakashima, R. Yokota, and N. Maruyama. A Task Parallel Implementation of Fast Multipole Methods. In *SC Companion*, pages 617–625, 2012.

S.-H. Teng. Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation. *SIAM Journal on Scientific Computing*, 19(2):635–656, 1998.

Michael S Warren and John K Salmon. A portable parallel particle program. *Computer Physics Communications*, 87(1):266–290, 1995.

R. Yokota. An FMM Based on Dual Tree Traversal for Many-Core Architectures. *Journal of Algorithms & Computational Technology*, 7(3):301–324, 2013.

Rio Yokota and Lorena A. Barba. Chapter 9 - treecode and fast multipole method for n-body simulation with CUDA. In *GPU Computing Gems Emerald Edition*, pages 113 – 132. Morgan Kaufmann, 2011.