Accelerating an FMM-Based Coulomb Solver with GPUs

Alberto Garcia Garcia, Andreas Beckmann, and Ivo Kabadshow

Abstract The simulation of long-range electrostatic interactions in huge particle ensembles is a vital issue in current scientific research. The Fast Multipole Method (FMM) is able to compute those Coulomb interactions with extraordinary speed and controlled precision. A key part of this method are its shifting operators, which usually exhibit $O(p^4)$ complexity. Some special rotation-based operators with $O(p^3)$ complexity can be used instead. However, they are still computationally expensive. Here we report on the parallelization of those operators that have been implemented for a GPU cluster to speed up the FMM calculations.

1 Introduction

The simulation of dynamical systems of N particles subject to physical potentials, such as gravitation or electrostatics, is a crucial issue in scientific research. This problem is commonly referred as the N-body problem, which has no analytical solution for $N > 3$. However, using an iterative numerical approach, the dynamical behavior of such systems can be simulated. Therefore, the total force exerted on each particle is computed at discrete time intervals, so that the velocities and positions of the particles can be updated.

A typical example is the simulation of a system of particles with electric charges q_i. The Coulomb force $\mathbf{F}_{ij}$ of a particle j with charge q_j acting on a particle i with charge q_i is defined by the following expression:

$$\mathbf{F}_{ij} = \frac{q_i q_j}{|\mathbf{r}_{ij}|^3}\mathbf{r}_{ij}\,, \tag{1}$$

A. Garcia Garcia • A. Beckmann • I. Kabadshow (✉)
Jülich Supercomputing Centre (JSC), Forschungszentrum Jülich GmbH, Jülich, Germany
e-mail: a.garcia@fz-juelich.de; a.beckmann@fz-juelich.de; i.kabadshow@fz-juelich.de

H.-J. Bungartz et al. (eds.), *Software for Exascale Computing – SPPEXA 2013-2015*, Lecture Notes in Computational Science and Engineering 113,
DOI 10.1007/978-3-319-40528-5_22

where $\mathbf{r}_{ij}$ is the distance vector between particles i and j. Given that, the total force $\mathbf{F}_i$ acting on each particle i can be expressed as the following summation:

$$\mathbf{F}_i = \sum_{j=1}^{N} \frac{q_i q_j}{|\mathbf{r}_{ij}|^3} \mathbf{r}_{ij} \quad (j \neq i)\,. \tag{2}$$

As we can observe, calculating the forces acting on each particle has a computational complexity of $O(N)$ since we have to compute all pairwise interactions of the current particle with the rest of the system. Therefore, a naive algorithm for computing all forces $\mathbf{F}_i$ exhibits $O(N^2)$ complexity. The update step has a complexity of $O(N)$ since computing the velocities from the forces just needs to iterate once over each particle. The same applies for the position update step. In this regard, the quadratic complexity may be negligible for a small number of particles, but interesting and useful simulations often involve huge particle ensembles, so the simulation will be considerably slowed down to a point in which it is non-viable to apply this kind of summation method. Fortunately, due to the increasing importance of N-body simulations for research purposes, fast summation methods have been developed throughout the latter years [1–3, 5, 7, 8].

In this work, we will focus on the Fast Multipole Method (FMM). The main goal is to develop a CUDA accelerated implementation of a rotation operator that is used during the FMM passes to reduce the computational complexity of the typical FMM mathematical operators, used for shifting and converting the multipole expansions, from $O(p^4)$ to $O(p^3)$. In contrast to other GPU implementations[13, 16], we focus our efforts on achieving good performance for a high multipole order ($p > 10$) required for MD simulations.

This document is structured as follows: Sect. 2 introduces the FMM, its core aspects and the role of the M2M operator as well as the functioning of the rotation-based operators. Section 3 describes the existing CPU/sequential implementation of the FMM $O(p^3)$ and $O(p^4)$ operators and sets baseline timings for all of them. Section 4 explains the changes in the application layout and the included abstraction layers to support the future GPU implementation. Section 5 shows the implementation details of the GPU-accelerated version using CUDA. In Sect. 6 we draw conclusions about this work and outline possible future improvements.

2 Theoretical Background

In this section we provide a brief description of the FMM, reviewing its core aspects and its mathematical foundations. We also briefly describe the role of the mathematical operators used for shifting. At last, we explain how the application of rotation-based operators to those expansions is capable to reduce the complexity of the aforementioned operators from $O(p^4)$ to $O(p^3)$.

2.1 The FMM Workflow

The FMM is a fast summation method which is able to provide an approximate solution to the calculation of forces, potentials or energies within a given precision goal, namely ΔE. The FMM developed at JSC is capable of automatically tuning [4] the FMM parameters for a given energy threshold ΔE. The method exhibits linear computational complexity $O(N)$, resulting from a sophisticated algorithmic structure. The core aspects of the FMM are: spatial grouping of particles, hierarchical space subdivision, multipole expansion of the charges and a special interaction scheme.

The main idea behind the FMM is based on the following intuitive property of the Coulomb and gravitational potential: the effect of particles close to the observation point (called target), on the target particle is dominant compared to the effect produced by remote particles. As opposed to a cutoff scheme, the FMM takes into account the effects of all particles no matter how remote. Cutoff methods have a $O(N)$ complexity, but ignore interactions beyond a cutoff completely.

Consider the particle distribution shown in Fig. 1, for which the remote interactions between two clusters shall be computed: target, with m particles, and source, with n particles. The FMM is based on the idea that a remote particle from a spatial cluster will have almost the same influence on the target particle as another one from the same cluster, given that the inter-cluster distance is large enough. The FMM therefore groups all particles in the remote cluster into a pseudo-particle. By doing this, the amount of interactions is effectively reduced to m. This grouping scheme is also used in reverse, by grouping the target cluster thus requiring n interactions. When grouping both source and target clusters, the computation reduces to a single however more complex interaction.

To implement spatial grouping, the simulation space is subdivided to generate particle groups. The FMM decomposes space recursively in cubic boxes, generating eight different child boxes from each parent box. This hierarchy of cubes is arranged in a tree, called *octree* of depth d. Figure 2 shows an example of this recursive subdivision visualized in a 2D plane.

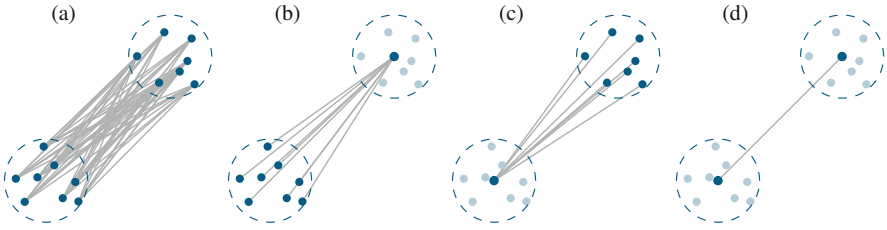

Fig. 1 From *left* to *right*: (**a**) Direct interactions of the particles of one cluster with all particles in the other cluster. (**b**) Interaction via source pseudo-particle. (**c**) Interaction via target pseudo-particle. (**d**) Interaction with both source and target pseudo-particles

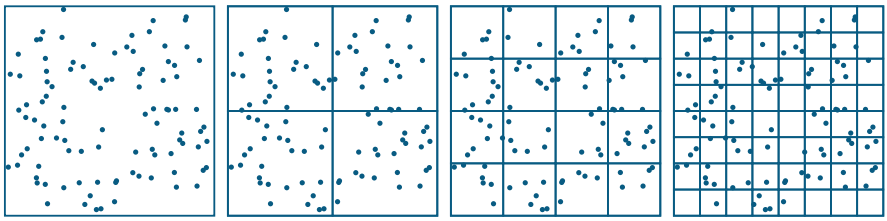

Fig. 2 Space subdivision using an octree. From *left* to *right*: Trees with depth $d = 0$, $d = 1$, $d = 2$, and $d = 3$

Given a certain separation criterion *ws*, the multipole order p and the depth of the tree d, the FMM consists of the following steps, called passes:

- **Pass 1:** Expand charges into spherical multipole moments ω_{lm} on the lowest level for each box, and translate multipole moments ω_{lm} of each box up the tree
- **Pass 2:** Transform remote multipole moments ω_{lm} into local moments μ_{lm} for each box on every level
- **Pass 3:** Translate local moments μ_{lm} down the tree towards the leaf nodes
- **Pass 4:** Compute far field contributions: potentials Φ_{FF}, forces $\mathbf{F}_{\text{FF}}$, and energy E_{FF} on the lowest level
- **Pass 5:** Compute near field contributions: potentials Φ_{NF}, forces $\mathbf{F}_{\text{NF}}$, and energy E_{NF} on the lowest level

This algorithm exhibits a linear computational complexity. Its derivation is beyond the scope of this work, and can be found in [11]. The first pass is performed by the P2M operator, which is often considered a preprocessing step, and the M2M operator, while the second one is done via the M2L operator and the third one with the L2L operator. This work focuses on the M2M operator. The extension to the remaining operators M2L and L2L is straightforward and can be implemented following the same strategies.

2.2 *Mathematical Operators*

As mentioned in Sect. 2.1, the FMM needs three fundamental mathematical operators during its workflow, namely M2M, M2L, and L2L. Those operators are responsible for shifting the multipole expansions up and down the tree levels, and also to convert remote multipole expansions to local ones at each level. We will briefly review the first operator to provide the context for the rotation-based operators, which is described in Sect. 2.3.

2.2.1 Multipole-to-Multipole (M2M) Operator

The M2M is a vertical operator which shifts the multipole coefficients up to higher levels of the tree structure. Each box of the 3D tree has eight child boxes in the next lower level. The M2M operator sums up all the moments of the multipole expansions of the child boxes at the center of the parent box. This operator is applied to each level up to the root of the tree. By doing this, each box on every level has a multipole expansion. This operator is applied in the first pass, and is also known as A.

From a mathematical perspective, each child multipole expansion ω^i at the center a_i of that child box i is shifted up to the center $a + b$ of its parent box (see Fig. 3). Equation (3) shows how the moments $\omega^i_{jk}(a_i)$ of each child multipole expansion are shifted by the A operator to produce the moments $\omega^i_{lm}(a_i + b_i)$ of the parent's expansion:

$$\omega^i_{lm}(a_i + b_i) = \sum_{j=0}^{l} \sum_{k=-j}^{j} A^{lm}_{jk}(b_i)\omega^i_{jk}(a_i) \ . \tag{3}$$

All the shifted moments of the eight child boxes are finally added up to conform the multipole expansion at the center of the parent box

$$\omega_{lm}(a + b) = \sum_{i=1}^{8} \omega^i_{lm}(a_i + b_i) \ . \tag{4}$$

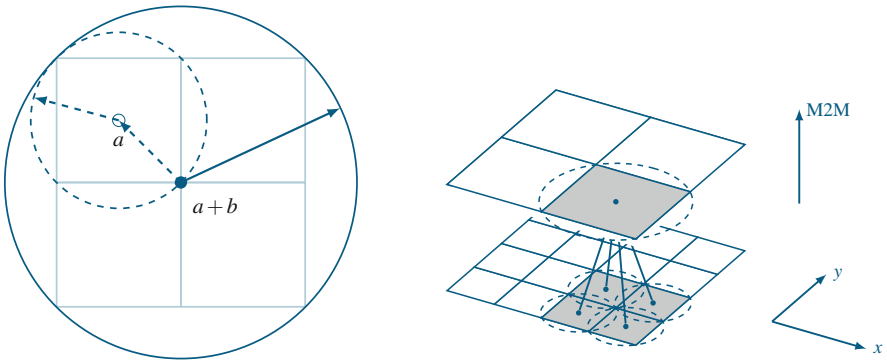

Fig. 3 The *left panel* shows the analytical domain of the M2M operator in a 2D tree. The centers (*blue dots*) of a sample child box and the parent are shown. The *right panel* depicts the functioning of the M2M operator for a 2D system. The operator has $O(p^4)$ complexity

2.3 Rotation-Based Operators

A set of more efficient operators with $O(p^3)$ computational complexity scaling were proposed by White and Head-Gordon [15]. The reduced complexity is achieved by rotating the multipole expansions so that the translations or shifts are performed along the quantization axis of the boxes (see Fig. 4). This reduces the 3D problem to a 1D one.

The multipole moments of an expansion with respect to a coordinate system which has been rotated twice, first by an angle ϕ about the z-axis and then by θ about the y-axis, can be expressed as a linear combination of the moments with respect to the original coordinate system. The rotated multipole expansion (see Fig. 4) can be expressed as ω'_{lm} as shown in Eq. (5):

$$\omega'_{l,m}(\theta,\phi) = \sum_{k=-l}^{l} \frac{\sqrt{(l-k)!(l+k)!}}{\sqrt{(l-m)!(l+m)!}} d_l^{m,k}(\theta) e^{ik\phi} \omega_{l,k} \ . \tag{5}$$

In the last equation, $d_l^{m,k}$ represents Wigner small d-rotation coefficients whose computation falls beyond the scope of this work. A detailed explanation and implementation on how to compute them can be found in [9]. The term $e^{ik\phi}$ represents a factor that is needed for each moment to compute the rotation. Usually, the operator will compute both terms on the fly, adding a prefactor to the $O(p^3)$ complexity of this operator. However, that prefactor can be removed by precomputing and reusing the constants.

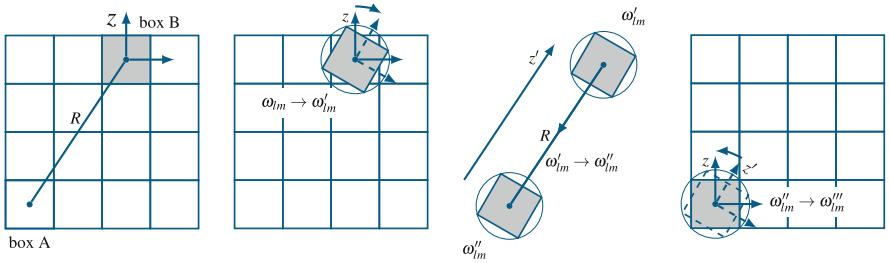

Fig. 4 The coordinate system of the box B is rotated to align it along the z'-axis defined by the quantization direction. The multipole expansion is translated into a multipole expansion around the center of A. The new multipole expansion is rotated back to the original coordinate system yielding ω'''_{lm}

3 Existing Implementation

In this section we will describe the existing C++ implementation of the FMM, which has been implemented within the GROMEX SPPEXA project. This project addresses the development, implementation, and optimization of a unified electrostatics algorithm that will account for realistic, dynamic ionization states (λ-dynamics) [6] and at the same time overcome scaling limitations on current architectures.

We will show benchmarks carried out to determine a baseline for future optimizations, i.e., the parallel implementation. In addition, this baseline will prove the effectiveness of the $O(p^3)$ operators.

From an application point of view, the FMM is implemented in a set of abstraction layers, each on top of another, with different responsibilities. By using a layered approach, the internal functionality of a layer can be changed and optimized at any time without having to worry about the other layers. This design provides flexibility, and it is implemented with the help of templates in the different layers (see Fig. 5).

As shown in the figure, the implementation is composed of four well distinguished layers: (1) the algorithm, (2) data structures, (3) allocator and (4) memory. The top layer contains the FMM logic itself, i.e., the implementations of the described passes. Here, we keep the focus on the M2M operator, for which templates allow us to choose between the $O(p^4)$ or $O(p^3)$ version.

Those implementations need data structures to store the information that is being processed. In this regard, the algorithm layer leverages to the data structures one. This layer contains the data types needed for the algorithm, e.g., coefficient matrices (ω), rotation matrices (R), and other simple data structures, including their internal

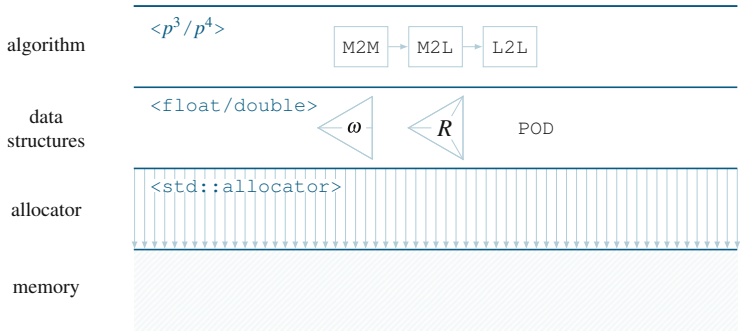

Fig. 5 Layout of the existing FMM implementation for CPUs. There are four different abstraction layers: the algorithm, the data structures, the allocator and the memory. The algorithm layer is templated to choose between the p^3 or p^4 operators. The data structures are also templated so that the underlying data type precision can be chosen. The allocator is templated as well, so that it can make use of custom or predefined memory allocators

logic. The templated design allows to choose the precision of the underlying data types.

The allocator layer enables the data structures to allocate memory for storing their information. The data structures delegate the memory allocation to an allocator performing the corresponding calls for allocating and deallocating memory. This layer is also templated, so any allocator can perform this task.

This existing CPU implementation was tested to establish a set of baseline performance results. A baseline helps to determine multiple facts: the actual effectiveness of the reduced complexity operators, the impact of the precomputed constants, and precision bounds. It will also serve as a starting point to compare the performance of the GPU implementation.

The benchmarks were carried out on the JUHYDRA cluster at the JSC, featuring an Intel Xeon E5-2650 CPU. Both versions of the M2M operator, $O(p^4)$ and $O(p^3)$, were compared. Our benchmarks are focused on this operator since it is the one that we decided to parallelize on the GPU as a starting point, given the fact that the optimizations performed over the M2M phase can be easily applied to the M2L and L2L ones. In addition, it can even be argued that porting the M2M phase to a GPU implementation is harder than the M2L one, due to less workload and parallelism. Note that the $O(p^3)$ operators employ some prefactors for the rotation steps. Those prefactors as well as the Wigner d-matrices are computationally expensive but can be precomputed to reduce the runtime. We tested both variants of the rotation-based operators: on the fly and precomputed. The benchmarks were carried out for both single and double precision floating point datatypes.

As seen in Fig. 6, the M2M $O(p^3)$ on the fly operator is even slower than the $O(p^4)$ version. However, when all the constants are precomputed the complexity reduction pays off because most of its prefactor penalty is removed. Nevertheless, there is still a small prefactor which makes it slower than its $O(p^4)$ counterpart when the order of poles is small. The single precision plot (Fig. 6 top) shows unexpected results in the interval for 10–20 multipoles. By taking a closer look, we can point out a significant runtime increase from multipole order 13 until order 18 for the $O(p^3)$ precomputed operator. The slope of the $O(p^4)$ operator also changes suddenly after $p = 15$. This behavior is caused by the limited precision of the `float` datatype. When a certain order of poles is requested, underflows in the multipole representation occur and the numbers fall in the denormalized range of the single precision type. Because of this, the denormalized exception handling mechanism of the FPU starts acting, thus increasing the execution time due to additional function calls. At a certain point, for instance $p = 18$ in the precomputed operator, the numbers drop to zero so no additional denormalized exception overhead is produced. That is the reason why the curve *stabilizes* after order 18. The `float` implementation achieves a 2.5× speedup before denormalization overhead starts at $p = 13$.

If we look at the double precision plot (Fig. 6 bottom), the unexpected slope does not occur since the double representation is able to handle the required precision. The runtime is reduced by one order of magnitude (13.8× speedup) when using order 50.

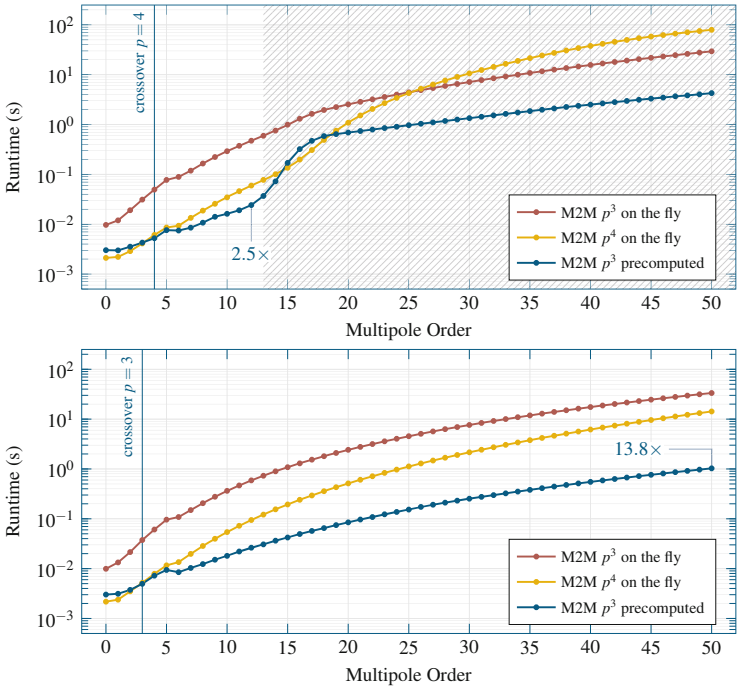

Fig. 6 Runtimes of a full M2M execution from the lowest level of an FMM tree with tree depth $d = 4$ to the highest one, i.e., 4096 M2M operator runs, shifting all boxes into a single one at the top level. Both M2M variants with $O(p^4)$ and $O(p^3)$ complexity were tested. The $O(p^3)$ operator was tested using non-precomputed constants which were calculated on the fly, and using those precomputed constants. The *upper panel* shows the timings using single precision floating point numbers and the *lower panel* shows those timings using double precision ones, varying the multipole order. For single-precision the speedup for the precomputed $O(p^3)$ compared to $O(p^4)$ was 2.5×, for double precision 13.8×. The kinks between multipole order four and six are only visible on the Sandy Bridge architecture, on Ivy Bridge the kinks are smaller, on Haswell the effect is not visible

In conclusion, a significant benefit is obtained by using the optimized operators. However, the complexity reduction implies a more sophisticated implementation and also a computationally more expensive prefactor, which should therefore be precomputed. These benchmarks establish a baseline for future improvements. In the following section, we will discuss the required steps prior to the CUDA-optimized implementation that will be deployed on a GPU.

4 Application Layout

Having a performance baseline using the existing CPU implementation, we start the code transformation into a CUDA-based one that can be deployed on GPU. The first steps consist of adapting the current application to support computations on the GPU. For that purpose, we need to allocate the data structures into the GPU memory so that they can be accessed directly by the CUDA kernels. Since we want to keep the changes in our codebase to a minimum, we leverage the application layout, previously described in Sect. 3. Those changes will heavily rely on templates and additional indirection layers. In this section, we describe the modifications applied to the aforementioned application layout to support efficient GPU execution.

4.1 Custom Allocator

The data structures need to be moved to GPU memory. This is achieved by explicit CUDA memory transfer calls whenever those data structures are needed. However, this approach will clutter the current application code since we need to include those explicit memory transfer operations in the algorithm layer, sabotaging the abstraction layer concept described in Sect. 3.

Modern NVIDIA GPUs provide a unified memory model that fits our requirements. Since all data structures make use of the allocator abstraction layer, we can just modify that layer without affecting the rest. In this way, we do not add any additional logic to the algorithm or the data structures. In addition, the allocator layer is templated so that we are flexible enough to choose between an allocator for CPU or GPU memory easily.

In this regard, we developed a custom CUDA managed allocator, which inherits from the `std::allocator` class and overrides the allocation and deallocation methods. It can be plugged in as a template parameter to our allocator abstraction layer.

In order to ease development, we decided to make use of unified memory despite its reduced efficiency when compared to other memory transfer operations, especially when data can be batched. Nevertheless, our current design provides complete control over the internal memory management mechanisms, so it can be easily extended to support other memory models or techniques such as overlapping memory transfers with computation.

4.2 Pool Allocator

The raw CUDA managed allocator has drawbacks if a considerable number of allocations has to be done. For big problem sizes bad allocations occur due to the limited amount of memory map areas a process can provide. The Linux kernel value

`vm.max_map_count` limited our allocations to 65,536. Since most of our data structures contain other nested ones, our implementation performs many allocation calls and for big problem sizes we eventually reach that limit.

There are several ways to solve this problem. In our case, we resorted to using a pool allocator with a reasonable chunk size to decrease the number of allocation calls that reach the operating system. This memory management scheme is integrated in our application as a new abstraction layer between the data structures and the actual allocator.

The pool allocator allocates chunks of a predefined size and then serves parts of those chunks to the allocation calls performed by the data structures. The improvement compared to the previous layout is twofold, (i) it decreases the execution time since each allocation call has a significant latency penalty, and (ii) allows to fully utilize the GPU memory without hitting the allocation calls limit.

Thanks to the decoupled design and the templated layers, introducing this middle level is straightforward. Neither algorithm logic nor data structures code has to be changed to include a new memory management strategy. We carried out a set of benchmarks that confirmed that adding this intermediate layer has no performance impact.

4.3 Merging the CPU and GPU Codebases

The pool allocator enables the application to efficiently deal with big problem sizes. However, two distinct implementations of the same routines exist for CPU and GPU architecture. As a result code cannot be reused at the algorithm level and if one implementation changes, the other has to be changed manually.

CPU kernels for the different operators and the rotation steps take references as input arguments by design. Additionally these kernels are usually implemented by a set of nested loops which iterate over all the elements of the coefficient matrix in a sequential manner. The GPU kernels make use of pointers to those data structures, and the loop starting points and strides are different since the threads will no longer iterate sequentially over them but rather choose the data elements to compute depending on their identifiers or positions in the block/grid.

To merge both implementations into a single codebase, the data structures are converted from references to pointers for the GPU kernel wrappers or launchers. The GPU kernels access the corresponding elements of the pointers to the data structures and call the operator or rotation kernels which make use of references. These operator and rotation kernels are used by both the CPU and the GPU. Figure 7 shows the final layout of our application with all the aforementioned layers.

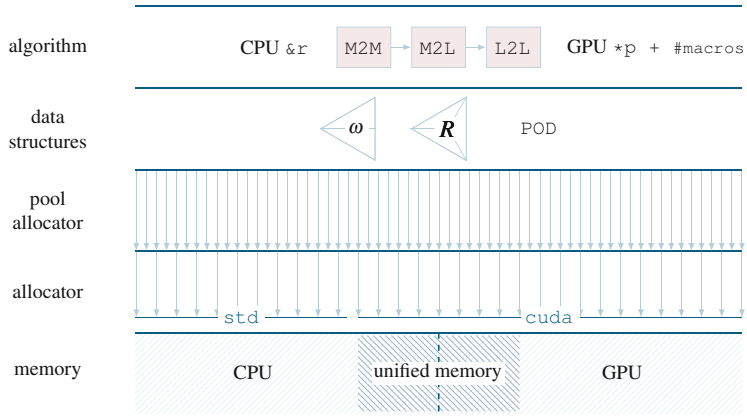

Fig. 7 Final application layout after merging the CPU and GPU logic for M2M, M2L, and L2L operators. The CPU uses references directly and the GPU launchers wrap them as pointers for the kernels. Also, preprocessor macros allow us to determine the loop starting points and strides, and specific features depending on the architecture

5 CUDA Implementation

As a starting point for the CUDA-optimized implementation, we focus on the M2M kernel. As we previously stated in Sect. 3, the optimizations performed over the M2M phase can be easily reused later for the M2L and L2L ones. Furthermore, it can even be argued that porting the M2M phase to a GPU implementation is harder than the M2L one, due to less floating point operations. Hence, an efficient implementation of M2M automatically enables an even better performance for the M2L operator, which has a significantly increased workload (see Fig. 8).

Now we will describe the parallelization of the $O(p^3)$ M2M operator, including both rotation steps, forwards and backwards. We will first focus on how to distribute the work to expose enough parallelism for the kernel functions to ensure a high GPU utilization. Then we will take an in-depth look at the different optimization strategies and CUDA techniques applied to each of the kernels. We will close showing the results of the accelerated operator and the speedup with respect to the CPU version.

5.1 Exposing Parallelism

A possible way to expose parallelism in a simple manner is to make each thread compute the whole operator for a single box, i.e., the rotation forward, M2M operator, and rotation backwards. This naive approach will spawn as many threads as boxes have to be processed. In the best case scenario we will have $(2^d)^3$ boxes in

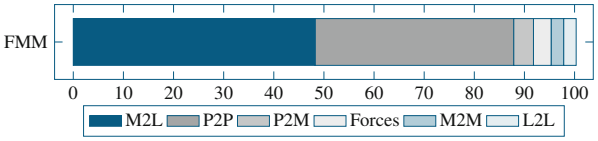

Fig. 8 Relative time distribution for the different passes of the FMM. Relative timings obtained after a full FMM run with 103k particles, $d = 4$, $p = 10$ and $ws = 1$. The CUDA parallelized versions of the $O(p^4)$ operators [12] were executed on a K40m

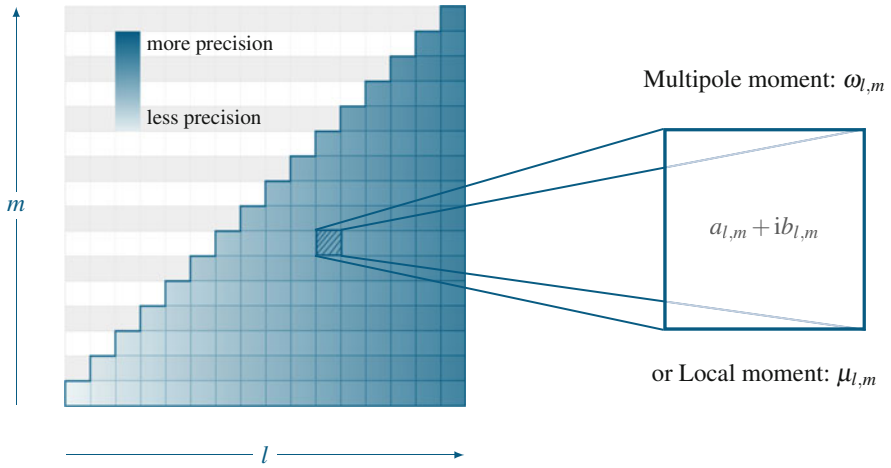

Fig. 9 Representation of the coefficient matrix datatype with an exemplary number of 15 poles. As the coefficient matrix grows, the precision increases, so a higher number of poles leads to more accurate simulations. Both axes are in the range $[0, p]$, this produces a coefficient matrix of $(p + 1)(p + 2)/2$ coefficients. The coefficients, which are represented as *squares* in the picture, are multipole or local moments depending on the type of the expansion. Each one of them holds a complex number

the lowest level of the tree. This means that for $d = 3$ we will launch 512 threads, and even for $d = 4$ only 4096 threads will be launched. Even for small block sizes, the grids will be composed of only a few blocks, preventing us to achieve a high GPU utilization.

Since there are not enough boxes to be processed, we have to take another approach to expose more parallelism. Before getting into any more detail, it is worth taking a look at the main data structure that is processed by the rotation and operator steps: the coefficient matrix. Figure 9 shows the representation of the coefficient matrix using only its upper part. It consists of a set of coefficients, representing the local or multipole moments, which are distributed in a triangular shape along the horizontal l and the vertical m axes. Each coefficient is represented by a complex number, and the rotation and operator steps usually iterate over all those coefficients to apply certain transformations (rotations, shifts, or translations). More parallelism can be exposed by assigning each warp (group of 32 threads which is the minimum

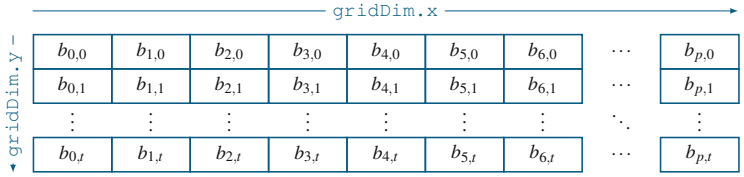

Fig. 10 Grid configuration. Each row is composed by $p + 1$ blocks b. The grid consists of t rows, with t being the total number of boxes minus one

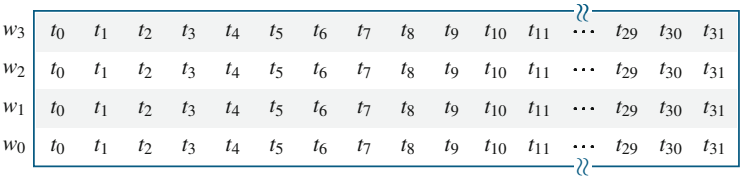

Fig. 11 Block configuration with 32×4 threads t. Each block is 32 threads wide and consists of four warps w of 32 threads each

unit processed in a SIMT fashion by a CUDA-capable device) the task of computing all the operations required for a certain coefficient.

Accordingly, we created the grid configuration shown in Fig. 10. Each block row is responsible of a full coefficient matrix. Since each box is represented by a coefficient matrix, the grid has one row per each box that has to be processed. Note that by using grid-strided loops [10], we can launch less blocks and distribute the work accordingly. The block configuration is shown in Fig. 11. Each consists of four warps of 32 threads thus creating a 32×4 2D structure.

Since each row of the grid is responsible of a full coefficient matrix, i.e., the `blockIdx.y` determines which coefficient matrix the block processes each block of the row is assigned to a certain column of the corresponding coefficient matrix. In other words, the `blockIdx.x` gets mapped to the l axis as shown in Fig. 12.

Once the blocks are mapped to the coefficient matrix, the next step does the same with the threads inside those blocks. Since we have groups of 32 threads inside each block which share the same y position, i.e., each warp has the same `threadIdx.y`, we can map the warps to individual coefficients or cells of the assigned column. This means that the `threadIdx.y` variable will be mapped to the m dimension of the coefficient matrix. Figure 13 shows the warp distribution for an arbitrary block. However, the distribution is not trivial since each column has a different height and after the fourth column there are more coefficients to process than warps in the threads.

To overcome this, the warps are reassigned to the remaining coefficients in a round-robin way. By doing this, warp zero will be always assigned to $m \in \{0, 4, 8, \ldots\}$, warp one to $m \in \{1, 5, 9, \ldots\}$, warp two to $m \in \{2, 6, 10, \ldots\}$ and warp three to $m \in \{3, 7, 11, \ldots\}$ taking into account the block configuration shown in Fig. 11. It is important to remark that, even considering that the warps will be

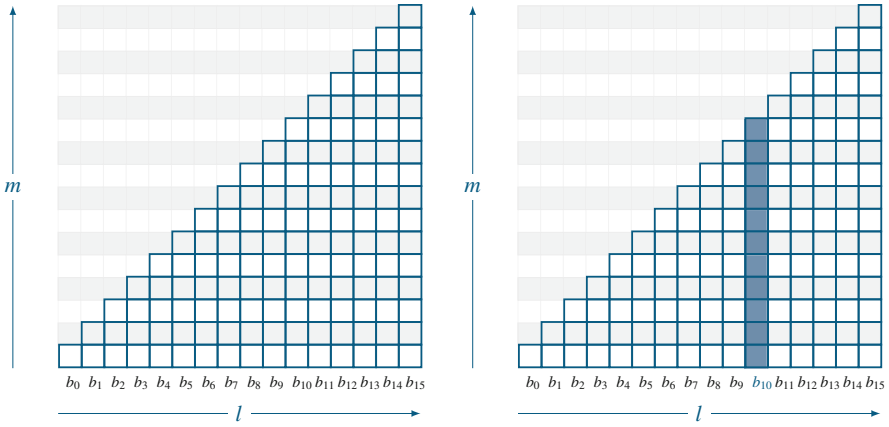

Fig. 12 Grid block row distribution. `blockIdx.x` is mapped to the l dimension of the corresponding coefficient matrix which is selected by the `blockIdx.y`, i.e., the y position of the block in the grid. The *left panel* shows the block mapping for an exemplary coefficient matrix with $p = 15$, so each grid row is composed of 16 blocks. The *right panel* shows an example of work assigned to a block. In this case the block b_{10} with `blockIdx.x` $= 10$ will have to compute all the coefficients of the highlighted column $l = 10$ of its corresponding coefficient matrix

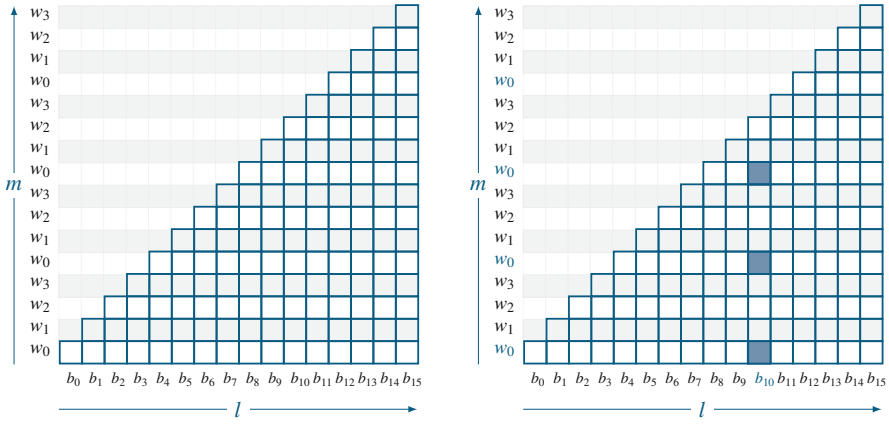

Fig. 13 Block warp distribution. Individual elements of the corresponding column of the coefficient matrix, depending on `threadIdx.y`, are mapped to warps in a round-robin fashion. The *left panel* shows the warp distribution in an exemplary coefficient matrix with $p = 15$. The *right panel* shows an example with the coefficients assigned to the first warp w_0 of a block b_{10} highlighted. That warp will compute the elements $(10, 0)$, $(10, 4)$, and $(10, 8)$. Although it would be assigned to the element $(10, 12)$ too, it will not compute it because it is outside the boundaries of the coefficient matrix ($m > l$)

theoretically assigned to a certain m, they will not process that coefficient if it is not part of the coefficient matrix. Basically, warps will only compute if $m \leq l$.

The next step is mapping the threads depending on their `threadIdx.x` value which identifies the 32 threads inside each warp. Depending on the step to be performed (rotation, operator or rotation backwards) different computations will be

carried out with the coefficient, either by using the elements of the same row or the ones from the same column.

The process of iterating over the coefficient matrix is implemented by two nested loops, one for the l dimension and another one for the m one. The computations carried out for each coefficient are implemented as another nested loop, namely k. Depending on the aforementioned possibilities, this loop will iterate from $k = 1$ to l, from $k = m$ to l, or from $k = 1$ to m. In the end, individual threads will perform the work of that inner loop, which means that `threadIdx.x` gets mapped to k.

With this parallelization scheme, sufficient parallelism is exposed, so that a low GPU occupancy does not limit the performance. For instance, with $d = 3$ and $p = 15$ the GPU will launch 983,040 threads for the previously shown grid configuration. For $d = 4$ and $p = 15$, $\approx 7.9M$ threads will be launched. For increasing problem sizes, we might get to a point where we can't launch all the blocks we need. However, the mappings are implemented using grid-strided loops, so we can support any problem size by launching an arbitrary number of blocks and reusing them in a scalable manner.

Nevertheless, this approach has also some drawbacks. Due to the shape of the coefficient matrix and the way the m loop is mapped, a workload unbalance is produced among warps of the same block. Figure 14 shows two examples of warps with unbalanced load. Ideally, all the warps will perform the same amount of work, otherwise the early finishers will have to wait for the long running threads to finish to deallocate their resources. Also, because of the pattern followed by the k loop,

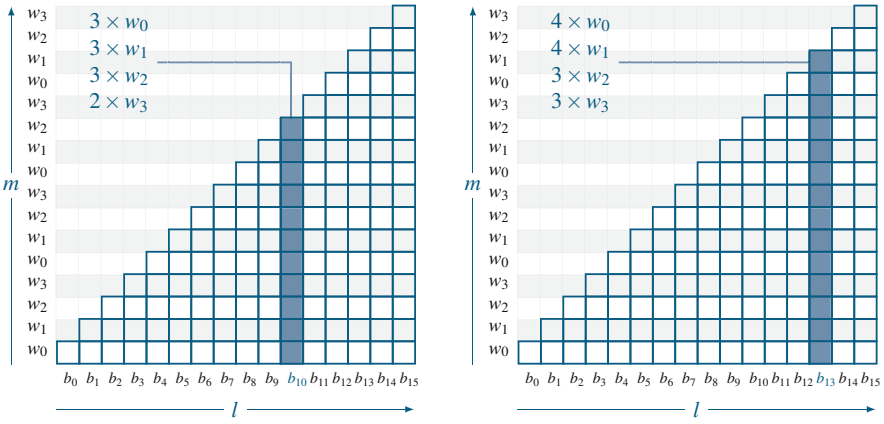

Fig. 14 Warp divergence due to different workload among the different warps of the same block. Blocks get scheduled and some of the warps finish earlier than others so they will be idle waiting for the others to end their workload. The resources allocated for the early finished warps are wasted since they will remain allocated until the longest running warp of the same block finishes. The *left panel* shows the workload for the different warps of block b_{10}, three warps will compute three coefficients each but the last warp will only compute two. The *right panel* shows another example of divergence in the block b_{13} since two warps will compute four coefficients each one and the other two will only calculate three each

not all the threads execute the same code path, so thread divergence may lead to performance degradation.

Despite the disadvantages, this strategy provides a starting point to start getting performance out of the GPU, although it can be improved to avoid the aforementioned pitfalls.

5.2 *Results*

We carried out a performance study to determine the improvement achieved by the CUDA-accelerated implementation. Figure 15 shows the results of that benchmark for single and double precision representations.

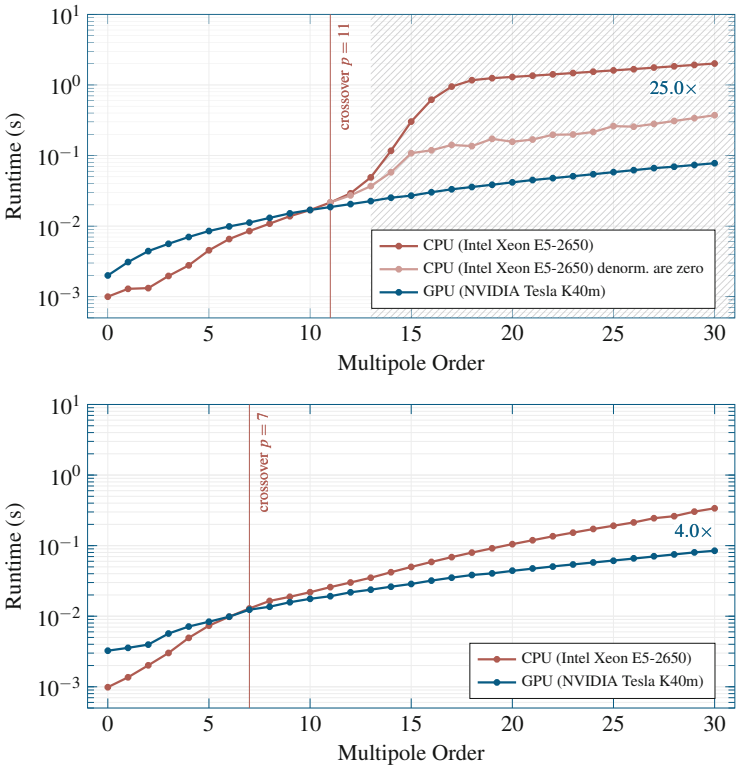

Fig. 15 Runtimes of full M2M $O(p^3)$ operator execution over the lowest level of an FMM tree with depth $d = 4$, i.e., 4096 M2M operator runs, shifting all boxes into a single one at the top level. Both CPU baseline implementation and GPU-CUDA-accelerated implementation are shown as a function of multipole order. The *upper panel* shows the results using single precision floating point numbers (with and without denormalization handling), the *lower panel* corresponds to double precision. All tests were executed on the JUHYDRA cluster at the JSC, the CPU tests ran on an Intel Xeon E5-2650 while the GPU ones used the NVIDIA Tesla K40m

For the floating point representation the same problem mentioned in Sect. 3 occurs again: the `float` datatype is not able to handle the required precision, leading to exception handling mechanisms of the CPU increasing the execution time. Currently, GPUs do not support denormalized numbers and truncate to zero immediately if an underflow occurs. For valid results, with $p < 10$, no gain is achieved by using the current GPU implementation.

The double precision benchmarks show that the extended representation is able to cope with the required precision. The crossover point is located at $p = 7$, from there the GPU implementation shows a faster execution time than the CPU one. A maximum speedup of $4.0\times$ is obtained at the biggest problem size tested, $p = 30$.

The results confirm that it is possible to improve the performance of the M2M operator by using a massively parallel device such as a GPU. However, a significant computational load is required to hide the costs of parallelism. Furthermore, there is still plenty of room for improvement, further optimizations, and architecture specific tuning. An in-depth profiling of the aforementioned kernels should provide guidance for improving the results.

6 Conclusion

In this work, we have shown how the rotation-based M2M operator of the FMM can be accelerated by executing it on a GPU using CUDA. In addition, we integrated both the CPU and GPU code into a single codebase using a flexible design, based on a set of abstraction layers to decouple responsibilities. The starting point was an existing FMM implementation pipeline with $O(p^3)$ and $O(p^4)$ operators. We analyzed the implementation by carrying out benchmarks to set a performance baseline for all the operators. This baseline helped to quantify the performance gain achieved by using the rotation-based operators.

We enhanced the code to make it able to execute on the CPU or on the GPU in a transparent manner using CUDA. For this purpose, a set of abstraction layers was introduced: (1) algorithms, (2) data structures, (3) pool allocator, and (4) memory allocator. We developed an accelerated version of the rotation-based M2M operator. The improvements made to that operator can be easily ported to the other ones. Our benchmarks show that the GPU-accelerated M2M operator runs up to four times faster than the highly optimized single-core CPU implementation when using double precision floating point representation.

The highlights of this work can be summarized as follows:

- a flexible application layout with a single codebase for the CPU/GPU implementation, based on a set of abstraction layers, atop of another:
  - an algorithm layer to hold the FMM logic
  - a data structures layer containing types, structures and their internal logic
  - a pool allocator layer for efficient memory management
  - an allocator layer for transparent memory space allocation

- a CUDA-accelerated version of the rotation-based M2M operator
  - flexible and scalable grid-strided loops
  - coalesced accesses to the data structures and fast warp reductions using CUB [14]
  - launch bounds to help the compiler optimize kernels
  - precomputed factors to save global memory round trips

Here we focused on building the abstraction layout and on accelerating the rotation-based M2M operator. The acceleration of the remaining operators M2L and L2L is straightforward since they share the same data representation and building blocks used for M2M. In addition, all the CUDA kernels can be further optimized to improve occupancy and reduce divergence. Furthermore, architecture specific tuning can be applied.

Acknowledgements This work is supported by the German Research Foundation (DFG) under the priority programme 1648 "Software for Exascale Computing—SPPEXA", project "GROMEX".

References

1. Appel, A.W.: An efficient program for many-body simulation. SIAM J. Sci. Stat. Comput. **6**(1), 85–103 (1985)
2. Barnes, J., Hut, P.: A hierarchical O (N log N) force-calculation algorithm. Nature **324**, 446–449 (1986)
3. Brandt, A.: Multi-level adaptive solutions to boundary-value problems. Math. Comput. **31**(138), 333–390 (1977)
4. Dachsel, H.: An error-controlled fast multipole method. J. Chem. Phys. **132**(11), 244102 (2009)
5. Darden, T., York, D., Pedersen, L.: Particle mesh Ewald: an N log(N) method for Ewald sums in large systems. J. Chem. Phys. **98**(12), 10089–10092 (1993)
6. Donnini, S., Ullmann, R.T., Groenhof, G., Grubmüller, H.: Charge-neutral constant ph molecular dynamics simulations using a parsimonious proton buffer. J. Chem. Theory Comput. **12**(3), 1040–1051 (2016)
7. Eastwood, J.W., Hockney, R.W., Lawrence, D.N.: P3M3DP-the three-dimensional periodic particle-particle/particle-mesh program. Comput. Phys. Commun. **19**(2), 215–261 (1980)
8. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. J. Comput. Phys. **73**(2), 325–348 (1987)
9. Gumerov, N.A., Duraiswami, R.: Recursive computation of spherical harmonic rotation coefficients of large degree. CoRR abs/1403.7698 (2014)
10. Harris, M.: CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops. http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/
11. Kabadshow, I.: Periodic boundary conditions and the error-controlled fast multipole method, vol. 11. Forschungszentrum Jülich (2012)
12. Kohnke, B., Kabadshow, I.: FMM goes GPU: a smooth trip or a bumpy ride? (2015), GPU Technology Conference

13. Lashuk, I., Chandramowlishwaran, A., Langston, H., Nguyen, T.A., Sampath, R., Shringarpure, A., Vuduc, R., Ying, L., Zorin, D., Biros, G.: A massively parallel adaptive fast multipole method on heterogeneous architectures. Commun. ACM **55**(5), 101–109 (2012)
14. Merrill, D.: CUB – collective software primitives (2013), GPU Technology Conference
15. White, C.A., Head-Gordon, M.: Rotating around the quartic angular momentum barrier in fast multipole method calculations. J. Chem. Phys. **105**(12), 5061–5067 (1996)
16. Yokota, R., Barba, L.: Treecode and fast multipole method for N-body simulation with CUDA. ArXiv e-prints (2010)