Optimization Techniques for Dimensionally Truncated Sparse Grids on Heterogeneous Systems

Andrei Deftu
Technische Universität München
andrei.deftu@in.tum.de

Alin Murarașu
Technische Universität München
murarasu@in.tum.de

***Abstract*—Given the existing heterogeneous processor landscape dominated by CPUs and GPUs, topics such as programming productivity and performance portability have become increasingly important. In this context, an important question refers to how can we develop optimization strategies that cover both CPUs and GPUs. We answer this for *fastsg*, a library that provides functionality for handling efficiently high-dimensional functions. As it can be employed for compressing and decompressing large-scale simulation data, it finds itself at the core of a computational steering application which serves us as test case. We describe our experience with implementing *fastsg*'s time critical routines for Intel CPUs and Nvidia Fermi GPUs. We show the differences and especially the similarities between our optimization strategies for the two architectures. With regard to our test case for which achieving high speedups is a "must" for real-time visualization, we report a speedup of up to 6.2x times compared to the state-of-the-art implementation of the sparse grid technique for GPUs.**

***Keywords*-sparse grids, GPU, library, optimizations, CUDA**

I. Introduction

The number of processor architectures has grown significantly in the recent years. We saw the appearance of accelerators such a GPUs as a response to the many obstacles met by processor architects, e.g. the frequency and memory walls. GPUs are application specific architectures, well suited for data parallelism, for code with regular memory access patterns such as those available in numerical or graphics applications. Compared to CPUs, GPUs contain simpler cores but compensate with their higher number (tens of cores compared to 4 - 8 for CPUs). The cores operate at a lower frequency (often 1 GHz against 2 - 3 GHz for CPUs) and are in-order, hiding the latency of the instruction pipeline by interleaving on each core the execution of thousands of threads.

An abstraction of GPUs is to see them as many-core architectures, with a large number of cores and with each core including large vector units allowing to compute up to 32 single precision floating point operations per cycle (64 flops if we count FMAD as 2 operations, i.e. a multiply and an addition). Nvidia GPUs can be programmed through the well known CUDA framework. Although the vector units are not directly visible in CUDA, the SIMD nature of GPUs reveals itself when applying various optimizations, e.g. for bank conflicts and memory access coalescing [1]. Viewing GPUs as vector processors is supported by a very efficient implementation of dense linear algebra operations on GPUs [2].

When reasoning about code transformations that improve performance on a wide range of architectures, we do not want to lose ourselves in the details of the programming models since this often results in lower programming productivity. A more abstract view focusing on hardware characteristics rather than programming models allows for more flexibility when adapting code to new architectures. For improved productivity and performance, our goal is to develop high-level code transformations that address characteristics common across different processors, namely CPUs and GPUs. In order to simplify the porting of an application, it is important to determine the similarities and differences between the old and the new architecture and map the optimizations accordingly. In defense of this statement, CPUs and GPUs both have vector units and array-of-structures to structure-of-arrays is a transformation that often is required by both. Given the current convergence trend between CPUs and GPUs, i.e. CPUs have doubled the width of the SIMD units from SSE to AVX and GPUs have been equiped with a 2 level cache memory [3], we expect that increasingly more transformations, e.g. for locality, to be shared between codes for CPUs and GPUs.

In order to support our ideas, we consider a computational steering application whose main goal is to provide real-time interaction with compressed simulation data [4]. The two hotspots of this application are the routines implementing two advanced algorithms for data compression and decompression. Data compression is in this case achieved through a numerical method called the *sparse grid technique* which is similar to wavelets to some extent. More exactly, sparse grids allow for a hierarchical and memory efficient representation of high-dimensional simulation data. Note that sparse grids must not be confused with sparse matrices mainly because of the high-dimensional settings in which sparse grids are used, e.g. 10-dimensional, as opposed to matrices which are 2-dimensional.

In order to reduce the response time of our computational steering application, it is important to leverage the latest hardware. However, restarting the optimization process with every major hardware release can be very counter-productive and therefore should be avoided. Instead, what we follow is to develop optimizations for loops and data layout at the same abstraction level at which we develop algorithms. The main conclusion supported by this paper is that most of the transformations proposed for our test application are applicable and beneficial to both CPUs and GPUs. Our optimization approach

is relies highly on an analysis of CPUs and GPUs based on similarities and differences, putting minimal emphasis on the programming model which in our view can actually complicate the porting process. Moreover, we believe that our study is important given the fact that increasingly more programming models are emerging and the open question is whether or not programmers can design optimizations portable across different types of processors, e.g. CPUs and GPUs.

The main contributions of our paper are as follows:

- We port for the first time the sparse grid library *fastsg* on GPUs and report a speedup of 28.3x compared to the optimized CPU version.
- We study the performance benefits of a set of loop transformations on CPUs and GPUs. We describe the challenges met along the way and their solutions.
- We report speedup numbers in the context of our computational steering application accelerate the state-of-the-art implementation up to a factor of 6.2x for compression and 1.5x for decompression on GPUs.

The rest of the paper is structured as follows. In Section II we place our work in the context of code optimizations and sparse grids, and relate to the published work. Section III highlights the similarities between CPUs and GPUs, allowing for a reuse of optimizations across architectures. Section IV introduces sparse grids and describes the core functionality of the sparse grid library *fastsg*. In Section V, we describe a set of loop transformations suitable for both architectures. Section VI shows our optimizations for CPUs and GPUs based on the aforementioned ideas. In Section VII, we present our results and show that despite using different architectures, similarities in terms of hardware characteristics have the potential to ease code porting. Finally, Section VIII concludes the paper.

II. Related Work

Sparse grids are a special type of grids known for their ability to address high-dimensional problems which suffer from the so-called "curse of dimensionality", i.e. the exponential dependency of the number of grid points on the number of dimensions. Sparse grid applications include solving PDEs, data mining and recently computational steering. For the detailed theory and applicability the reader is referred to [5].

A data structure with minimal memory consumption and efficient algorithms for sparse grids are described in [6]. Further functionality, i.e. for dimensionally truncated sparse grids, and an efficient implementation on CPUs is covered by [7]. Dimensionally truncated sparse grids give the flexibility to treat dimensions differently, by allowing more or less discretization points per dimension, and thus to control the refinement level of the grid on a per dimension basis. Compared to previous work, we propose loop transformations that allows us to efficiently port dimensionally truncated sparse grids on GPUs. To the best of the authors' knowledge this is the first GPU implementation of dimensionally truncated sparse grids. Moreover, emphasis is put on exposing a set of loop transformations that increase the performance on both CPUs and GPUs.

A set of techniques for efficient programming of heterogeneous systems for dense linear algebra are presented in [8]. The authors show there how the computation can be divided across different processor types in order to fully use the system, but do not address the problem of programming CPUs and GPUs based on leveraging their similarities.

Frameworks such as MapCG [9] which offers source code level portability based on the MapReduce paradigm or StarPU [10] which provides an uniform execution model through different task scheduling strategies are proof that efforts have been made to offload computational workload across multicore processors and accelerators. Our work is inspired by applications such as these. However, whereas their approach is to provide a high-level model for easier programming, our focus is on code structure and loop transformations to achieve optimal performance in a portable manner.

III. CPUs and GPUs

Heterogeneous systems containing CPUs and accelerators allow us to reach higher computational speeds while keeping power consumption at acceptable levels. The latest developments show a trend towards an unification in terms of hardware features of both architectures. Instead of focusing only on the classic way to increase the speed of CPUs by raising the frequency with each new release, the manufacturers are now trying to exploit other directions as well, such as vectorization or many-cores. An eloquent example is Intel's Many Integrated Core Architecture (MIC) [11] which is announced to include 512-bit wide vector registers and more than 50 cores per chip (in Knights Corner product line). On the other hand, GPUs manufacturers started to include CPU-like features in their products. Fermi [3] was the first GPU generation from Nvidia featuring a cache coherent memory and today all devices tend to have a cache hierarchy similar to the one on CPUs.

A. GPU Architecture

GPUs are heavily optimized for fast arithmetic operations and in consequence have many dedicated floating point units. Therefore, not all applications are suitable for running on GPUs, or at least not as efficient as on CPUs, but only those that exhibit regular memory access pattern and require a great amount of computational power.

The CUDA architecture from Nvidia is based on *Streaming Multiprocessors* (SM), which are SIMD cores with the role of creating, scheduling and executing concurrent threads. One such multiprocessor is composed of several lanes, or *Scalar Processors* (SP), and has a dedicated shared memory for fast access by the threads running on it. The thread-execution manager automatically schedules threads on processors without the need from the programmer for writing explicit code. The execution is done in groups of 32 threads, called *warps*, such that every instruction is broadcast synchronously to all the active threads in a *warp*. The degree of parallelism is high when all these 32 threads execute the same instruction and very low when branching causes thread divergence.

Each thread has a small local private memory. Threads are grouped in *blocks*. Each *block* has a shared memory accessible by all threads in the *block*. Designed for graphics processing, the cards are also equipped with dedicated, read-only, on-chip memories: constant cache for low latency and texture cache for high bandwidth. Finally, there is also the global memory which has the largest capacity, but is the slowest. Starting with the Fermi architecture, a finer level of control is possible for configuring how the allocation of these memories is done, giving the programmer more flexibility, but at the cost of exposing complexities of the underlying architecture.

B. GPU Programming

While SIMD code on CPUs can be written using assembly, a high-level language like C (with intrinsic functions), libraries such as Intel Array Building Blocks or just relying on a compiler with auto-vectorizing capabilities, Nvidia GPUs are usually programmed through their dedicated framework, called CUDA. CUDA allows the programmer to define functions, or *kernels*, to be executed in parallel by many threads on different processors. When the *kernels* are launched on the GPU, a given topology must be specified. Threads are grouped in *thread blocks* which are laid in a grid. Each *thread block* has a unique 3d index which gives its position in the grid, while each thread has a 3d index identifying the thread in a *block*. This allows uni-, bi-, and three-dimensional topologies, the right configuration depending on the problem to solve. Threads in a *block* are scheduled to be run on a single SM. This means synchronization is only possible among threads in the same *block*, using the *__syncthreads* barrier. These threads can also share data through the shared memory of the SM.

Optimizations on GPUs address mostly thread scheduling and memory access. The most important ones in the first category are maximizing the number of active threads per SM by choosing the right *kernel* topology and minimizing the number of divergent branches in a *warp* that could lead to a serial execution. Regarding the memory hierarchy, it is preferable to use on-chip memories whenever possible, to reduce the number of bank conflicts when accessing the shared memory, and to reduce the number of memory transactions to main memory through coalescing. For a detailed list of optimizations we refer the reader to [1].

IV. Computational Steering Using Sparse Grids

Our application is the visualization of compressed, high-dimensional data resulting from simulations [4]. It is well known that managing data coming from high-dimensional functions has an exponential complexity on the number of dimensions. Therefore, we compress the data using the sparse grid technique in order to reduce its size and decompress it afterwards for real-time visualization. For this purpose we use the *fastsg* library [7].

The main idea behind sparse grids is that we can approximate a D-dimensional function $f : \Omega \rightarrow \mathbb{R}$, where $\Omega = [0,1]^D$, by discretizing Ω and representing f as a weighted sum of *basis functions*. If we consider *levels* $\bar{l} = (l_1, \ldots, l_D)$ and *index* $\bar{i} = (i_1, \ldots, i_D)$ vectors in $\mathbb{N}^D$, then each basis function will be centered at points $\bar{x}_{\bar{l},\bar{i}} = (x_{l_1,i_1}, \ldots, x_{l_D,i_D}) \in \mathbb{Q}^D$, for which $x_{l,i} = i \cdot 2^{-l}$, $i \in \{1, \ldots, 2^l - 1\}$, and i odd. This means that each grid point $\bar{x}_{\bar{l},\bar{i}}$ can be uniquely identified by the pair $(\bar{l}, \bar{i})$. Thus,

$$f \approx \sum_{\bar{x}_{\bar{l},\bar{i}}} \alpha_{\bar{l},\bar{i}} \cdot \phi_{\bar{l},\bar{i}}$$

where $\alpha_{\bar{l},\bar{i}}$ is the weight, or *hierarchical coefficient*, and $\phi_{\bar{l},\bar{i}}$ is the basis function centered at grid point $\bar{x}_{\bar{l},\bar{i}}$. In our case, $\phi_{\bar{l},\bar{i}}$ is obtained by multiplying D one-dimensional functions $\phi_{l,i} = h(2^l x - i)$, where h is the standard hat function $h(x) = max(1 - |x|, 0)$:

$$\phi_{\bar{l},\bar{i}}(\bar{x}) = \prod_{t=1}^{D} \phi_{l,i}(x_t).$$

The *truncated sparse grid*, $\Omega_a \subseteq \Omega_r$, identified by a given constraint vector $\underline{c}$, is the set of points

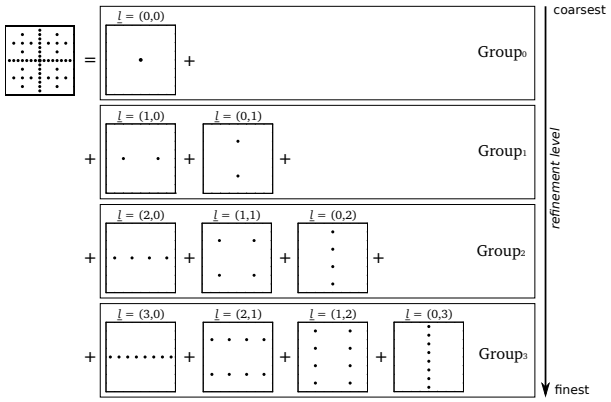
$$\Omega_a := \{\bar{x}_{\bar{l},\bar{i}} : |\bar{l}|_1 \leq L + D - 1, l_t \leq c_t, t \in \{1, \ldots, D\}\}.$$

If $c_t = L, \forall 1 \leq t \leq D$ then Ω_a becomes a regular sparse grid, i.e. regular sparse grids are special types of dimensionally truncated sparse grids.

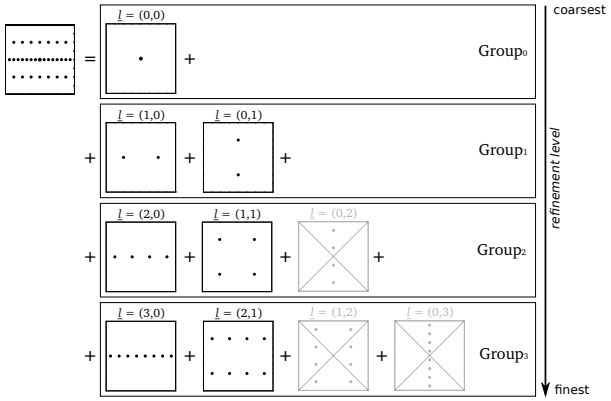
Compressing a general function represented on a full grid is done by selecting only the function values at grid points also contained in the sparse grid Ω_a. Many points are excluded from the full grid using the Ω_a discretization, thus resulting in a form of lossy compression. Sparse grids theory is based on the fact that the excluded points have a reduced contribution to the approximation [5]. The consequence is that for sufficiently smooth functions, removing the points does not significantly deteriorate the accuracy compared to full grids. Compression is reduced to computing the hierarchical coefficients α, process called *hierarchization*. Decompression, also called *evaluation* or *interpolation*, refers to evaluating the sparse grid approximation anywhere inside the domain by summing up the contributions of the basis functions averaged by their hierarchical coefficients. This also enables us to interpolate at points for which we do not have values from simulation. Hence, it can provide hints on the simulation outside the initial data. In our case, decompression is based on the sparse grid technique described in [5].

A. Compression

It is important that sparse grids should not be confused with sparse matrices. They differ in both functionality and computational behavior. As mentioned, sparse grids allow us to approximate multi-dimensional functions. The method is somehow related to wavelet transformation technique [12], where computing and grouping of the coefficients leads to a multi-scale representation. Similar to wavelet compression, we also have here a hierarchical representation which corresponds to different scales or levels of detail. The contribution of the coefficients reduces as this level increases.



(a) Regular sparse grid.



(b) Dimensionally truncated sparse grid.

Fig. 1: Decomposition of 2D sparse grids.

Sparse grid compression has 4 input parameters: number of dimensions D, refinement level L, precision P (single or double precision), and truncation T (dimensionally truncated or regular). T gives the possibility of reducing the computational effort when using functions where not all dimensions carry equal weight. Therefore, instead of storing all regular grids, we can exclude some of them when the information on those dimensions is not critical (see Fig. 1b). In our case, this translates to restricting the components of the $\bar{l}$ vector. An analogy can be made with image compression. Usually images are not squares, but rectangular in shape, having the width twice the size of their height. Going back to our sparse grids domain, we can thus use a refinement level in one dimension (width) equal to 2x the refinement level in the other dimension (height).

Fig. 1a shows how a sparse grid can be represented as a sequence of regular grids, grouped by their norm into several refinement levels, which will be simply referred to as *groups*. The grids in one *group* will be called *blocks*[1]. Instead of using the traditional approach of storing the grids in trees and hash-tables, we are using a bijection based data structure with minimum memory footprint [6]. All hierarchical coefficients are efficiently stored in a 1-dimensional array, by mapping $(\bar{l}, \bar{i})$ pairs to consecutive integer indices.

[1]Grid *blocks* should not be confused with CUDA thread *blocks*.

Compressing the multi-dimensional function translates to computing the hierarchical coefficients, in a traditionally recursive manner by collecting the contributions of all points and evaluating the basis functions in each dimension. With the above data structure, we can use a non-recursive GPU friendly algorithm for compression. Although this algorithm is integer-bound, the next section shows that it can be made memory-bound by applying different loop transformation.

B. Decompression

Interpolating at a given D-dimensional point means traversing the set of regular grids and computing the contribution of each one. For each regular grid, a D-linear basis function is built in $\mathcal{O}(D)$ and evaluated at that point. Interpolating at one point uses exactly one value from each regular grid for scaling the basis function. We can now analyse the influence of input parameters on the performance of interpolation, visible especially on GPUs.

Sparse grid decompression has 5 input parameters: number of dimensions D, refinement level L, number of interpolation points N, precision P, and truncation T. D increases the computational intensity, i.e. the ratio between the on-chip computation time and off-chip communication time. On GPU, a large D causes an increased consumption of shared memory per thread reducing the benefits of multithreading. A large L decreases the computational intensity since the size of the regular grids increases exponentially, i.e. from 2^0 to 2^{L-1}.

V. Loop Transformations in *fastsg*

The majority of optimizations applied to numerical codes focus on loops since these are the places where most of the execution time is spent. In general, we expect the compiler to handle loop transformations. An overview of loop transformations performed by the compiler is presented in [13]. We cover in our paper only a subset of those optimizations relevant for our discussion and our routines. Specifically, we focus on optimizations that improve the latency for memory access by exploiting the deep memory hierarchy of modern processors, optimizations that decrease the number of instructions executed and help vectorization, and those that speed up arithmetic operations with integers.

A. Memory Locality

The first category includes transformations which improve memory locality. As pointed out in Section III, GPUs have started to borrow caches and scratchpad memories from CPUs, making them susceptible to a plethora of new optimizations. *Loop tiling* is a loop reordering transformation primarily used to improve cache reuse by dividing the iteration space into tiles. The transformation is highly dependent on the memory hierarchy. For GPUs the tile can be chosen such that it fits into the cache of the microprocessor, allowing the assignment of one or more tiles to each microprocessor. Aggressive tiling is done for all memory levels, including registers, L1 and L2 caches, etc. The complexity of caches complicates compiler's

task to produce efficient loop tiling transformations. An example of application for this kind of transformation on GPUs is stencil codes [14], which was first developed for CPUs. In the same category of loop transformations that enhance locality, there is also *loop interchange*, which exchanges the position of two loops so that the access to memory is improved, i.e. fewer cache and TLB misses. Sometimes, because the number of iterations of a loop is not known at compile time, the compiler cannot determine the best permutation of loops. Complex data dependencies resulting from complex array subscripts can also limit the applicability of this optimization since the compiler needs to prove that a given optimization preserves the semantics of the unoptimized code. A study about automatic loop interchange on GPUs is presented in [15].

B. Vectorization

In the recent period, *vectorization* has been gaining increased attention from developers. The current generation of processors from Intel, Sandy Bridge, contains 8-lane SIMD units (AVX), providing two times more throughput than the previous 4-lane SIMD units (SSE). In theory, automatic speedup can be achieved through vectorization performed by compilers. However, there are several factors that limit compilers' ability to vectorize a loop: data dependencies, aliasing, indexing arrays with complex expressions, non-sequential access patterns, etc. Furthermore, vectorization is usually applied after other loop transformations. Hence, if the compiler does not reorder the loops, the innermost loop cannot be vectorized. We refer the reader to [16] for a complete vectorization guide on Intel processors and conclude that manual intervention is sometimes required to cope with such inhibitors.

Among loop transformations that help compilers, there are loop unrolling, loop unroll-and-jam, and software pipelining. *Loop unrolling* replicates the body of a loop a number of times referred to as the unrolling factor. This optimization also reduces the loop overhead and can improve the register, cache, and TLB locality. Another advantage is that more possibilities for reordering instructions are available, thus resulting in a better utilization of the instruction pipeline. The downside is an increase in size of the generated code which puts more pressure on the instruction cache. *Loop unroll-and-jam* is similar to unroll in the sense that it copies the body of one of the next inner loops some number of times equal to the unroll-and-jam factor. The generated inner loops are then fused together, or jammed. This transformation can lead to more independent instructions in the inner loop. Both loop unrolling and loop unroll-and-jam pave the way for instruction level parallelism, but on GPUs this also translates to a higher register usage, which in turn leads to a reduced microprocessor occupancy. These optimizations combined with various other techniques proved successful when porting some kernels from linear algebra, like matrix multiplication [2] or symmetric matrix vector product (SYMV) [17]. *Software pipelining* implies reordering instructions from different iterations of a loop such that independent instructions are executed in sequence. It is often used in combination with loop unrolling.

C. Integer Operations

Strength reduction involves replacing computationally expensive operations with arithmetically equivalent, but less expensive ones. In our particular case, instead of using divisions inside the loop body, we precompute the inverses outside the loop and do multiplications instead. Another important optimization in this category is *loop invariant code motion*. This is a data-flow based loop transformation that moves outside a loop the computation whose result does not modify between iterations. The obvious benefit is a decrease in the number of instructions executed.

VI. Optimization Strategies

The computational core of *fastsg* resides in *hierarchize* and *evaluate* routines which do the compression and decompression, respectively. Their full description is available in [6]. We extend the set of loop transformations from [7] and present them in the context of CPUs and GPUs. As before, we group these optimizations into 3 categories based on their end purpose: memory locality, vectorization, and operations with integers, synthesized in Table I.

A. Compression

We start with a basic version of *hierarchization* algorithm (Alg. 1) for GPU where we loop with t variable over all dimensions, with g variable over all *groups* and then we call the kernel procedure. Each warp on the GPU computes the coefficients for one *block* of the sparse grid in order to achieve a better cache locality and to reduce the thread divergence within warps. Because the coefficients for one *block* are stored contiguously in memory, we also want for these values to be accessed simultaneously by a warp. Therefore, in loop j, each thread within a warp starts reading at index equal to its lane and then subsequently every element found at a distance multiple of 32 (the number of threads within a warp). In lines 8-9 we use the bijection to move from the global index of the grid point to $(\bar{l}, \bar{i})$, the *levels* and respectively *index* vectors representing the coordinates of one point in the grid. Since they are frequently used, we store them in the shared memory. The bijection functions use many integer-only operations. The routines *left* and *right* from lines 10-11 return the left and respectively right parents (*blocks* that the current one depends on) of a point in dimension t, while *agp2idx* returns the index where point $(\bar{l}, \bar{i})$ is stored in the 1d representation of the sparse grid, *asg1d*. The statements in lines 12-13 exhibit a non-sequential memory access pattern.

We now improve this basic algorithm with the goal of exploiting our data structure and *block*-level decomposition. We start with optimizations for integer operations. Since within a *block*, $\bar{l}$ vector has the same value, optimization *inv1* does a loop invariant code motion and shifts the computation of $\bar{l}$ outside the j loop.

Next, *inv2* computes before the innermost loop the indices inside *asg1d* of all $g-1$ parents corresponding to each *block* of norm g. What is left to compute is the index within the *block*. This is done inside j loop using $\bar{i}$.

TABLE I: Summary of optimizations.

Category	Abbreviation	Optimizations
Memory locality	*ichg1*, *ichg2*	loop interchange, loop tiling
Vectorization	*vec1*	loop unroll-and-jam, software pipelining
Integers	*inv1*, *inv2*, *inv3*, *inv4*	loop invariant code motion
	sred1	strength reduction, loop invariant code motion

Algorithm 1 Compression on GPU

1: **for** $t = 1$ to D **do**
2: **for** $g = L$ downto 1 **do**
3: HIERARCHIZEKERNEL(t, g)
4: **procedure** HIERARCHIZEKERNEL(t, g)
5: **for** $j \leftarrow$ BLOCKSTART$(g) + lane$;
6: $j <$ BLOCKEND(g); $j \leftarrow j + 32$ **do**
7: **if** $lane = 0$ **then**
8: $\bar{l} \leftarrow idx2l(j)$
9: $\bar{i} \leftarrow idx2i(j)$
10: $(\bar{ll}, \bar{il}) \leftarrow$ left$(\bar{l}, \bar{i}, t)$
11: $(\bar{lr}, \bar{ir}) \leftarrow$ right$(\bar{l}, \bar{i}, t)$
12: $v1 \leftarrow$ *asg1d*[agp2idx$(\bar{ll}, \bar{il})$]
13: $v2 \leftarrow$ *asg1d*[agp2idx$(\bar{lr}, \bar{ir})$]
14: *asg1d*$[j] \leftarrow$ *asg1d*$[j] - (v1 + v2) \cdot 0.5$

inv3 starts from the observation that $(\bar{ll}, \bar{il})$ and $(\bar{lr}, \bar{ir})$, the *levels* and *index* vectors of the left and right parents, differ from $\bar{l}$ and $\bar{i}$ of the current point only in one dimension. Therefore, we can reduce the complexity for computing the index within the *block* from $\mathcal{O}(D)$ to $\mathcal{O}(1)$.

In the innermost loop over the current *block*, now there are only the conversions $(\bar{l}(t), \bar{i}(t)) \rightarrow (\bar{ll}(t), \bar{il}(t))$ and $(\bar{l}(t), \bar{i}(t))$ $\rightarrow$ $(\bar{lr}(t), \bar{ir}(t))$, which have a complexity of $\mathcal{O}(\bar{l}(t))$. *inv4* precomputes the results of these conversions before the kernel launch and stores them in an array. At a cost of higher memory footprint, we thus obtain $\mathcal{O}(1)$ for the innermost loop.

Lastly, in order to improve memory locality and the data access patterns, *ichg1* optimization does a loop interchange by moving the loop over the *groups* from line 2 inside the kernel. In this way, one warp will not only process one *block*, but also all its parents.

B. Decompression

We move now to the algorithm for *evaluation* (Alg. 2). The coordinates of all N points to be evaluated are stored contiguously in a matrix $x[N][D]$ which is processed in chunks, each warp computing one chunk of size w in loop j. The memory access pattern for the warps is the same as for *hierarchization* algorithm where each thread computes points stored at indices multiple of 32 starting from the corresponding lane. The g and b loops traverse *groups* and *blocks*, respectively. *idx23* is used to index the beginning of the current *block* in *asg1d*, while *idx1* indexes the current point in this *block*. In line 13 we compute the coefficient position, while in line 16 we evaluate the basis function in that point. Line 17 updates the current value of the coefficient exploiting the *block*-level structure and exhibiting a non-sequential memory access pattern.

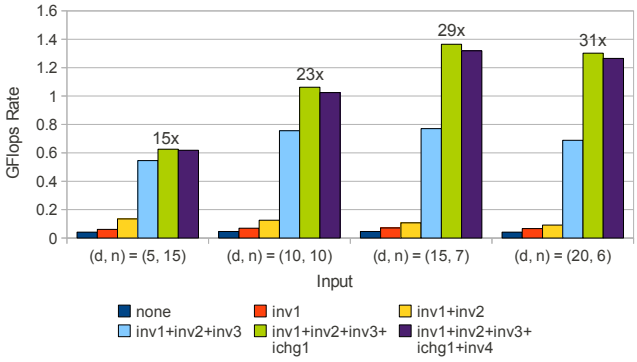
We start with optimizations for a better vectorization. In the CPU version, *vec1* does loop unroll-and-jam, loop tiling, and vectorizes the innermost loop t. Software pipelining can be done in the unrolled loop, but x is accessed with stride-D instead of stride-1. Therefore, we change the layout of x such that $x[j][t] \rightarrow \tilde{x}[(j/m2) \cdot D + t][j \bmod m2]$, where *m2* is the size of the tile. Now, a stride-1 access on $\tilde{x}$ is possible which makes SSE vectorization efficient. For GPUs, we note that the loop over dimensions in line 11 has a non-contiguous access pattern for a warp because at each read instruction, threads access elements found at D distance. *vec1'* improves this by using a technique similar to the one described before for both algorithms to increase the throughput through memory coalescing by threads within a warp. More specific, *vec1'* transposes the matrix x such that instead of storing the points horizontally, one point per row, it stores them vertically in $\tilde{x}[(N/32 + 1) \cdot D][32]$. Thus, all threads in a warp access one row in $\tilde{x}$ for each iteration over dimensions. This has a direct impact on GPU performance by coalescing the accesses to global memory and by avoiding memory bank conflicts.

Regarding memory locality, *ichg2* does a loop interchange by moving the loop over interpolation points from line 2 inside the loop over *blocks* from line 6. In the case of GPUs, we have two variations. *ichg2'* corresponds to *ichg2*, while *ichg2"* additionally moves the loops over *groups* and *blocks* outside the kernel. Their goal is to improve read accesses by having the innermost loop as tight as possible over the block of memory.

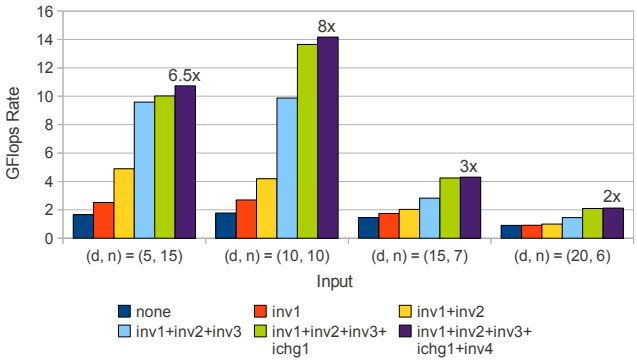
The last optimization is a combination of strength reduction and loop invariant code motion introduced in order to speed up arithmetic operations. *sred1* precomputes all divisions in lines 12-16 outside the innermost loop and replaces them with multiplications with the inverse. In addition,

Algorithm 2 Decompression on GPU

1: **procedure** EVALUATEKERNEL(w)
2: **for** $j \leftarrow$ CHUNKSTART$(w) + lane$;
3: $j <$ CHUNKEND(w); $j \leftarrow j + 32$ **do**
4: *idx23* $\leftarrow 0$
5: **for** $g = 1$ to L **do**
6: **for** $b = 1$ to $a(D, g)$ **do**
7: *idx1* $\leftarrow 0$
8: $p \leftarrow 1$
9: **if** $threadIdx.x = 0$ **then**
10: Compute $\bar{l}$ for which pos$(\bar{l}) = b$
11: **for** $t = 1$ to D **do**
12: $div \leftarrow 2^{-\bar{l}[t]}$
13: *idx1* $\leftarrow$ *idx1* $\cdot\, 2^{\bar{l}[t]} + \lfloor x[j][t]/div \rfloor$
14: $left \leftarrow \lfloor x[j][t]/div \rfloor \cdot div$
15: $right \leftarrow left + div$
16: $p \leftarrow p \cdot basis(left, right, x[j][t])$
17: $r[j] \leftarrow r[j] +$ *asg1d*[*idx1* + *idx23*] $\cdot\, p$
18: *idx23* $\leftarrow$ *idx23* $+ 2^g$

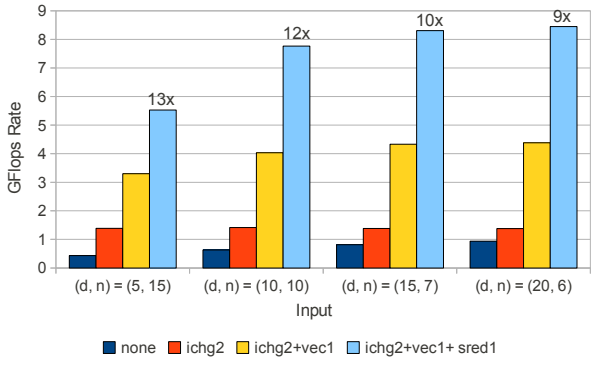


(a) CPU

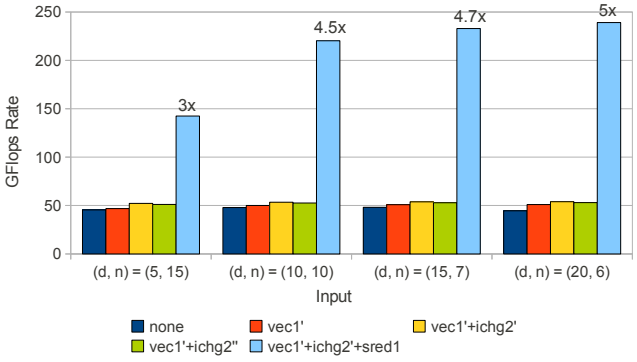


(b) GPU

Fig. 2: Impact of optimizations on the *hierarchization* algorithm.



(a) CPU



(b) GPU

Fig. 3: Impact of optimizations on *evaluation* algorithm.

$idx1 \leftarrow idx1 \cdot 2^{\bar{l}[t]} + \lfloor x[j][t]/div \rfloor$ from line 13 is replaced with $idx1 \leftarrow idx1 + \lfloor x[j][t]/2^{\bar{l}[t]} \rfloor \cdot 2^{prefix\_sums[t+1]}$, where $prefix\_sums[p] = \sum_{p=t}^{D-1} \bar{l}[p]$. This makes indexing a normal reduction and increases the instruction level parallelism.

VII. Evaluation

In this section we analyse the impact of the optimizations from Section VI on both CPU and GPU implementations.

Our test system is a quad-core Intel Core i7-920 running at 2.67 GHz with Hyper-threading enabled. For compiling the sequential CPU version we use *gcc 4.6* with flags "-O3 -march=native -funroll-loops -fargument-noalias". The GPU card is GeForce GTX 480 from Nvidia (Fermi architecture) with 1.5 GB of total memory and 15 multiprocessors, each with 32 lanes operating at a clock speed of 1.4 GHz. For programming the GPU, we started from the CPU code base and implemented the algorithms and optimizations with the established CUDA framework. The device supports CUDA Capability 2.0 which, among other features, includes the possibility for the same on-chip memory to be used for both L1 cache and shared memory. It can be configured as either 48 KB of L1 cache and 16 KB of shared memory, or vice versa. We use the first option for the *evaluation* algorithm in order to cache the accesses to local and global memories and the temporary register spills. Compilation is done by *Cuda Toolkit 4.1* with flags "-arch=sm_20".

We present the impact of our optimizations by measuring the GFlops rate[2] and comparing with the reference baseline case where no optimizations are applied. We start with the results obtained for the *hierarchization* algorithm, depicted in Fig. 2. The tests are done using different configurations, as displayed on the X-axis, where d is the number of dimensions and n is the number of refinement levels. When applying all optimizations, a speedup of up to 31x is obtained for CPU and up to 8x for GPU. With the increase in the number of dimensions and the decrease of the sparse grid size, the speedup for GPU tends to decrease. The explanation is that the effects of most our optimizations, e.g. improving caching and memory access patterns, are visible when a high degree of parallelism is achieved, as is the case for large grids. It is also interesting to notice that *inv4* brings a drop of performance when applied on top of the other optimizations for CPU, although for GPU it brings a speedup. This means that if the grids are small, then it is better to actually do the computations instead of doing memory lookups, which may result in cache misses. Giving this empirical optimization, a heterogeneous system could activate or deactivate it at runtime based on the architecture and compiler used. Lastly, we show that a speedup of 17x is obtained for the most performant GPU version compared to the corresponding CPU-based one and a speedup of 6.2x compared to the state-of-the-art GPU version [6].

In the *evaluation* algorithm, we use 10^4 for CPU and 10^6

[2]GFlops rate for the GPU version also includes the transfer time between GPU and main memories.

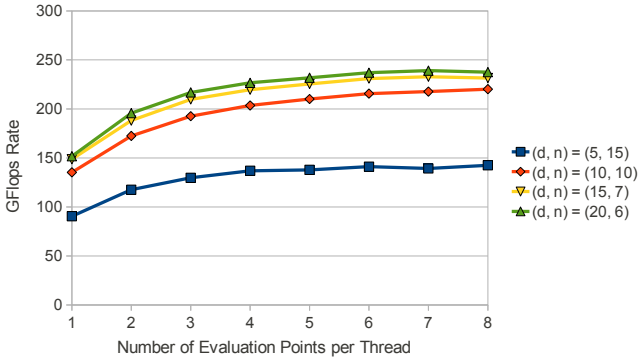


Fig. 4: Varying computational workload for *evaluation*.

for GPU random points, uniformly distributed in the function domain. We choose a higher number for GPU because the optimizations are visible when processing a large amount of data. Here we make two empirical observations. First, since the GFlops rate for CPU remains constant when increasing the number of interpolation points, both CPU and GPU versions use comparable input data. Second, we use 8 chunk sizes, i.e. number of interpolation points, for each warp: 32, 64, 96, 128, 160, 192, 224, and 256, but only choose the best in each case when comparing the optimizations. For reference, Fig. 4 shows how increasing this workload influences the results when applying all optimizations in different input configurations. Although the pattern suggests an improvement with the increase of thread workload, the results are not conclusive, which motivates for a careful tuning of the algorithm before running.

The results of our tests are presented in Fig. 3. Compared to the CPU versions, the optimizations for GPU are not that effective. The main reason for this is that, giving the high memory requirement for storing the interpolation points, we use the GPU global memory. Although caching is performed, we believe that a higher throughput could be obtained if we could have stored the results in shared memory. The optimization which brings a clear speedup is *sred1*, showing that division instructions are still very expensive on GPUs, fact also emphasized by [1]. The GPU version is 28.3x faster than the CPU one which confirms that sparse grids interpolation technique is cache friendly and easily parallelizable using our data structure. Compared to the state-of-the-art GPU version [6], a speedup of 1.5x is obtained for this algorithm.

VIII. Conclusion

In this paper we presented a series of loop transformations portable across multicore processors and accelerators, applied in the context of a computational steering application. We argued that giving the architectural similitude of CPUs and GPUs seen both as vector processors with different vector units sizes and roughly same cache hierarchy, some optimizations, at least conceptually, should be the same. With these concepts in mind, we proposed an extensive set of optimizations for both CPUs and GPUs that provide a speedup of up to 12.8x and 5.2x respectively.

In addition, we presented the first implementation of *hierarchization* and *interpolation* algorithms for dimensionally truncated sparse grids on GPUs and proved that our data structure and associated algorithms are easily parallelizable using these portable loop transformations.

We see the convergence of CPUs and GPUs both in terms of hardware features and programming languages. On heterogeneous systems OpenCL [18] is slowly making its way, but it is not as mature as CUDA and remains similarly complex. OpenACC [19] is a new standard with a lot of promise which focuses on parallelization of loop nests. These types of frameworks will lift the burden of focusing on architectural differences when programming, while at the same time incorporating common transformations and using auto-tuning techniques.

Acknowledgment

This publication is based on work supported by Award No. UK-C0020, made by King Abdullah University of Science and Technology (KAUST).

References

[1] NVIDIA, *CUDA Programming Guide 4.0*, 2011.
[2] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC'08*, pp. 31:1–31:11.
[3] NVIDIA, *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, White paper, 2009.
[4] D. Butnaru, D. Pflüger, and H.-J. Bungartz, "Towards high-dimensional computational steering of precomputed simulation data using sparse grids," *Procedia Computer Science*, vol. 4, no. 0, pp. 56 – 65, 2011.
[5] H.-J. Bungartz and M. Griebel, "Sparse grids," *Acta Numerica*, vol. 13, pp. 147–269, 2004.
[6] A. Murarasu, J. Weidendorfer, G. Buse, D. Butnaru, and D. Pflüger, "Compact data structure and scalable algorithms for the sparse grid technique," in *PPoPP'2011*, pp. 25–34.
[7] A. Murarasu, G. Buse, J. Weidendorfer, D. Pflüger, and A. Bode, "fastsg: A fast routines library for sparse grids," in *ICCS 2012*, Jun. 2012.
[8] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid gpu accelerated manycore systems," *Parallel Comput.*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010.
[9] C.-T. Hong, D.-H. Chen, Y.-B. Chen, W.-G. Chen, W.-M. Zheng, and H.-B. Lin, "Providing source code level portability between cpu and gpu with mapcg," *Journal of Computer Science and Technology*, vol. 27, pp. 42–56, 2012.
[10] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.
[11] Intel, *Intel Many Integrated Core (Intel MIC) Architecture*, 2011, ISC'11 Demos and Performance Description.
[12] S. G. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674–693, 1989.
[13] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994.
[14] V. Volkov, "Programming inverse memory hierarchy: case of stencils on GPUs," in *ParCFD 2010*.
[15] A. Leung, O. Lhoták, and G. Lashari, "Automatic parallelization for graphics processing units," in *PPPJ 2009*. New York, NY, USA: ACM, pp. 91–100.
[16] Intel, *A Guide to Vectorization with Intel C++ Compilers*, 2011.
[17] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra, "Optimizing symmetric dense matrix-vector multiplication on gpus," in *SC'2011*, pp. 6:1–6:10.
[18] Khronos, *OpenCL website*, http://www.khronos.org/opencl.
[19] OpenACC.org, *OpenACC website*, http://www.openacc-standard.org.