Workload Balancing on Heterogeneous Systems: A Case Study of Sparse Grid Interpolation

Alin Muraraşu, Josef Weidendorfer, and Arndt Bode

Technische Universität München
{murarasu,weidendo,bode}@in.tum.de

Abstract. Multi-core parallelism and accelerators are becoming common features of today's computer systems, as they allow for computational power without sacrificing energy efficiency. Due to heterogeneity, tuning for each type of compute unit and adequate load balancing is essential. This paper proposes static and dynamic solutions for load balancing in the context of an application for visualizing high-dimensional simulation data. The application relies on the sparse grid technique for data compression. Its performance critical part is the interpolation routine used for decompression. Results show that our load balancing scheme allows for an efficient acceleration of interpolation on heterogeneous systems containing multi-core CPUs and GPUs.

1 Introduction

Heterogeneous systems containing CPUs and accelerators allow us to reach higher computational speeds while keeping power consumption at acceptable levels. The most common accelerators nowadays, GPUs, are very different compared to state-of-the-art general-purpose CPUs. While CPUs incorporate large caches and complex logic for out-of-order execution, branch prediction, and speculation, GPUs contain significantly more floating point units. They have in-order cores which hide pipeline stalls through interleaved multithreading, e.g. allowing up to 1536 concurrent threads per core[1]. Garland et al. [1] refer to CPUs as latency oriented processors with complex techniques used for extracting Instruction Level Parallelism (ILP) from sequential programs. In contrast, GPUs are throughput oriented, containing a large number of cores (e.g. 16) with wide SIMD units (e.g. 32 lanes), making them ideal architectures for vectorizable codes. All applications can be run on CPUs but only a subset can be ported to or deliver good performance on GPUs, making them special purpose processors. In the following, we refer to GPUs and CPUs as processors, but of different type.

To support all kinds of heterogeneous systems in a *portable* way, we need to make sure that even for GPU-friendly code parts, there is a fallback to execute on CPU, as we also want to best exploit systems with powerful CPU parts. For that, multiple code versions of the same function have to be provided. For multi-core CPUs, OpenMP [2] is the de facto programming model. Nvidia GPUs on

[1] In Nvidia terminology a core is called Streaming Multi-Processor.

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part II, LNCS 7156, pp. 345–354, 2012.

the other hand are best programmed using CUDA [3]. OpenCL [4] targets both CPUs and GPUs. Still, for optimal performance, multiple versions are essential to target the different hardware characteristics. Another crucial part for efficient programming of heterogeneous systems is adequate workload distributing.

The main contribution of this paper consists of proposed solutions for load balancing in the context of the decompression of high-dimensional data compressed using the sparse grid technique [5]. This technique allows for an efficient storage of high-dimensional functions. Sparse grid interpolation (or decompression) is the performance critical part. For realizing load balancing, we employ a dynamic strategy in which the computation is decomposed at runtime into tasks of a given size (the *grain size*) which are grabbed for execution by the CPU and the GPU. We compare this strategy to a static approach, where the load distribution is done at the beginning of the computation, according to the computational power of the heterogeneous components. By this, we show that our interpolation runs efficiently on heterogeneous systems. To the best of our knowledge, this is the first implementation of sparse grid interpolation that optimally combines code tuned for multi-core CPUs and Nvidia GPUs.

2 Related Work

Our work is complementary to the one described in [6]. There, space and time efficient algorithms for the sparse grid technique are proposed. We use these algorithms as basis for our implementation of sparse grid interpolation for CPU and GPU. It is worth mentioning that in [6] the focus is on porting the sparse grid technique to GPUs. While the GPU code is executed, the CPUs are idle. Instead our goal is to avoid having idle processors and to further improve performance.

Similar to our approach, MAGMA [7] exploits heterogeneous systems by providing efficient routines for linear algebra. StarPU [8] is a framework that simplifies the programming of heterogeneous systems. Programs are decomposed into StarPU tasks (bundles of multi-version functions for every processor type) with according task dependencies, and automatically mapped to available processors (CPU / GPU). StarPU implements a distributed shared memory (DSM) over the CPU and the GPU memory via software controlled coherence. This allows for automatic data transfers to / from the GPU memory. Parameters exposed by StarPU to programmers are e.g. task size, task priority, and schedulers.

3 Optimizing Programs for Heterogeneous Computing

Programming the CPU and the GPU is inherently different. Multi-core CPUs are programmed using threads through pthreads or OpenMP. For GPU programming, CUDA is also based on threads, but there are differences. For synchronization, CUDA only provides barriers within thread groups running on the same GPU core, and atomic operations. For performance, the architectural details of GPUs have to be considered. Maximizing the number of threads running concurrently on the GPU, coalescing accesses to global memory, eliminating bank

conflicts, minimizing the number of branches, and utilizing the various memories appropriately (global, shared, texture, constant) are important GPU optimizations. In contrast, CPU optimizations include cache blocking and vectorization.

When programming heterogeneous systems with CPUs and GPUs, we can use an off-loading approach, as used in systems with co-processors for specific tasks. We determine a mapping between each function and the type of processor on which its execution time is minimal. As each function is executed by one type of processor, there is a risk for idle compute resources[2]. The solution is to move from off-loading to full function distribution. For this, we provide multi-version functions. We design them such that the CPU and the GPU cooperate for computing each function. Since this approach allows for a full utilization of a heterogeneous system, we focus on it in the rest of the paper.

Multiple versions of the same function must be orchestrated by an upper layer responsible for balancing the workload, either statically or dynamically. A *static* approach distributes the workload according to the computational speed of the processors. An initially determined distribution does not change during the execution of the function. In contrast, *dynamic* load balancing allows for changing the workload distribution after the computation has been started. It can be triggered by overloaded (sender initiated) or underloaded (receiver initiated) resources, can be executed in centralized or decentralized manner, and results in direct rebalancing (e.g. work stealing) or in repartitioning the data mapped to compute resources for the next iteration of the computation on that data. [9] provides a good overview of dynamic load balancing strategies. A typical dynamic strategy is receiver initiated load balancing of pieces of work which are not pre-mapped to given compute resources, but only distributed shortly before execution (also known as self-scheduling). This is also found in the OpenMP dynamic scheduling strategy for parallel for-loops. We call this the *dynamic task based approach*. The computation is decomposed into tasks which are inserted into a global queue. From there, the tasks are extracted by worker threads. Often, the tasks have dependencies, making the extraction more time-consuming. Variations use multiple queues or scheduling strategies based on work stealing, on greedy algorithms or algorithms that predict distribution costs. For heterogeneous systems, the worker threads invoke according versions of a function on the CPU or the GPU.

While the dynamic task based approach adapts implicitly to different machines, different input parameters, and external system load, there is an overhead for task queue management and distribution. Especially, the task size, called *grain size* in the following, influences that overhead. If it is too large, load balancing may not be achievable. If it is too small, the overhead may dominate and destroy any speedup. In contrast, the overhead of static balancing is minimal. Obviously, there is no grain size problem, but it has to adapt to function input parameters and machine type. If the workload depends not only on parameters such as data size, but on data values, static balancing is not feasible.

[2] Note that our objective is minimal execution time, not minimal energy consumption.

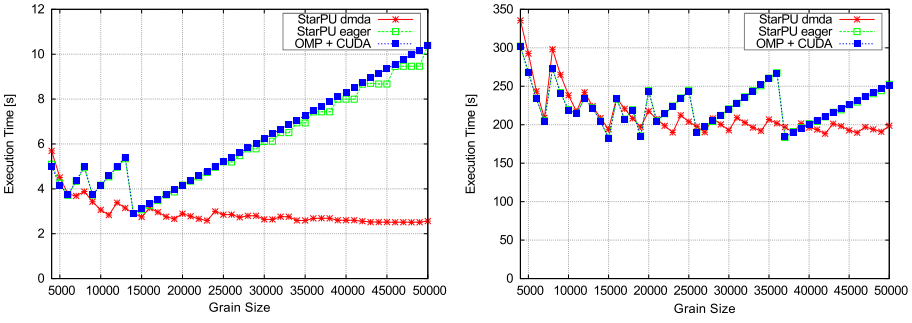


Fig. 1. Grain size impact. D/L/N = $6/12/5 \times 10^5$ (left), $20/6/3 \times 10^6$ (right)

We now focus on the importance of the grain size in the dynamic task based approach. In addition to the previous general remarks, a highly tuned CPU version of a function performs the best for a task size that matches or is a multiple of the tile size used for cache blocking. On the GPU, the task size should match or be a multiple of the maximum number of active threads. This would ensure full utilization of the GPU cores, of the SIMD units, and of multithreading. For sparse grid interpolation, we developed an according first-come first-served scheduler strategy using OpenMP and CUDA (OMP + CUDA). Moreover, we implemented our application with StarPU, using various schedulers available there. Fig. 1 shows the performance of interpolation for different grain sizes with different input parameters: number of dimensions (D), refinement level (L), and number of interpolations (N). The measurements are done using a Quad-core Nehalem and an Nvidia GTX480. Note that the optimal grain size depends on these parameters, especially for StarPU eager and our OMP + CUDA scheduler. The dmda scheduler assigns tasks based on a performance model that considers execution history and PCIe transfer overheads. For more details we refer to [8].

4 Sparse Grid Interpolation

Our application is the visualization of compressed, high-dimensional data resulting from simulations [10]. Decompression is in our case a form of interpolation based on the sparse grid technique described in [5]. Fig. 2 depicts an example of 5d data, i.e. velocity field, obtained from simulating the lid driven cavity for different Reynolds numbers (Re). The velocity of cavity's upper wall can also be transformed into a parameter, making this a 6d problem. For a high number of dimensions, managing the data can pose serious challenges. Therefore, we compress the data using the sparse grid technique in order to reduce its size and we decompress it afterwards for real-time visualization. This technique 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.

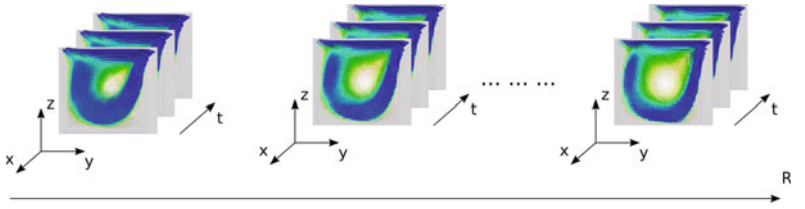
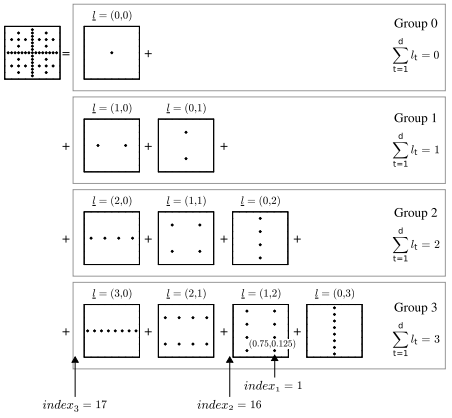


Fig. 2. 5d (x, y, z, t, Re) data from a CFD simulation

Sparse grid interpolation has 5 input parameters: the number of dimensions (D), the refinement level (L), the number of interpolations (N), the precision (P) (single or double precision), and the adaptivity (A) (adaptive or regular). In this paper we concentrate on the first 3, these being the most important as they can take a wide range of values. Fig. 3 (left) shows that a sparse grid can be represented as a sequence of regular grids [6]. Using this storage scheme, we can explain the interpolation and the impact of the inputs on performance. Interpolating (Fig. 3 (right)) at a given D-dimensional point means traversing the set of regular grids and computing the contribution of each regular grid on the result. For each regular grid a D-linear basis function ($\mathcal{O}(D)$) is built and evaluated at the point. Interpolating at one point uses exactly one value from each regular grid for scaling the basis function.

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}. We can see this in Fig. 3 (left) for L = 3 (regular grids of sizes between 2^0 and 2^3). As only one regular grid value is used per interpolation, only a small percentage of the compressed data transferred over PCIe to the GPU is actually used for computation. N is proportional to the computational intensity, i.e the more interpolations we perform, the more worthwhile is the data transfer over PCIe.

Our versions of interpolation are based on the iterative algorithm from [6]. The CPU version is optimized for best use of caches and vector units. Our GPU implementation includes the following optimizations: coalesced memory accesses, use of shared memory, no bank conflicts, etc. Having these two versions of interpolation, we combine them so that all the processors in a heterogeneous system simultaneously work on interpolation. In general, on the systems where we measured the performance of interpolation, the GPU was faster than the CPU. But, since our goal is performance portability, it makes sense to consider the situation in which the GPU is not faster than the multi-core CPUs available in the system. This can be the case for instance with Intel's Sandy Bridge processors which have a SIMD unit [11] (256 bit AVX) twice as wide as the previous generation, Nehalem (128 bit SSE). The parallelization of sparse grid interpolation is based on distributing the points for interpolation among threads.




```
Interpolation routine

Input:
   - set of points at which we interpolate, IP
   - sparse grid, SG

Output:
   - interpolation results, o

Algorithm:
   for every point p in IP
      init. result of interp. at p, o[p] = 0
      for every group g in SG
         for every regular grid r in g
            c = contribution of r to interp. at p
            o[p] = o[p] + c
```

Fig. 3. Left: 2d sparse grid decomposed as a sequence of regular grids. Group l (l = $0 \ldots 3$) contains C^l_{D+l-1} regular grids of size 2^l. D expands the groups horizontally while L expands them vertically. Right: simplified interpolation.

5 Interpolation and Heterogeneous Computing

Having two optimized versions for CPUs and GPUs, we want to interpolate simultaneously on all the processors of a heterogeneous system. For this, workload balancing is essential. This section details our approaches for load balancing.

5.1 Dynamic Task Based Load Balancing

Dynamic load balancing offers a natural way to allow the fastest processor to grab a number of tasks proportional with its speed. But, failing to determine the optimal task can seriously reduce the performance. For maximum performance, we treat the grain size as a tunable parameter. Finding its optimal value can be difficult when it is influenced by the input parameters of the application (Fig. 1). This is the case with sparse grid interpolation.

Each combination of values for the inputs can determine a different optimal value for the grain size. This complicates the process of tuning this parameter. The 3d space determined by D, L, and N (or 5d if we add P and A) can make the search for the optimal grain size very time-consuming or even impractical. To reduce the time spent by the search we use a performance model that returns in an acceptable amount of time an approximation of the execution time for each combination of values for the inputs. Our model is based on the following system of linear equations:

$$T'_{cpu}(w) = n_{cpu} \cdot t_{cpu}(w) \tag{1}$$

$$T'_{gpu}(w) = n_{gpu} \cdot t_{gpu}(w) + t_{pcie} \tag{2}$$

$$(c_{cpu} \cdot n_{cpu} + c_{gpu} \cdot n_{gpu}) \cdot w = N \tag{3}$$

$$T'_{cpu}(w) = T'_{gpu}(w) \tag{4}$$

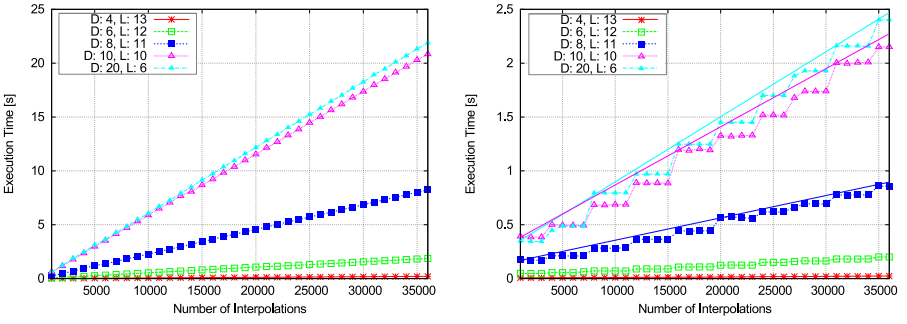


Fig. 4. Left: Execution time on the CPU as a function of workload. Dependence is linear. Right: Execution time on the GPU as a function of workload. The steps result from: large number of cores, wide SIMD units, and multithreading.

$T'_{cpu}(w)$, $T'_{gpu}(w)$, n_{cpu}, and n_{gpu} are the unknowns. The first equation builds the approximation T'_{cpu} of the execution time on the CPU, T_{cpu}, as the product between the number of tasks grabbed by a worker thread (n_{cpu}) and the duration of a task as a function of workload ($t_{cpu}(w)$), i.e. the workload is equivalent to the number of points at which we interpolate. Similarly, the approximation of the execution time on the GPU, T'_{gpu}, is the sum between the duration of all tasks executed on the GPU ($n_{gpu} \cdot t_{gpu}(w)$) and the one-time overhead (t_{pcie}) caused by transferring the compressed data over PCIe. The third equation means that the total workload equals the sum of the workload handled by CPUs and the workload handled by GPUs. c_{cpu} is the number of CPU cores or CPU worker threads and n_{cpu} is the number of interpolations allocated to a core. c_{gpu} is the number of GPUs and n_{gpu} is the number of interpolations per GPU. Finally, the fourth equation expresses that the CPU and the GPU finish at the same time.

We now have to find good approximations (linear or piecewise) for the $t_{cpu}(w)$ and the $t_{gpu}(w)$ functions depicted in Fig. 4. These can be considered cheap operations since the definition domain of these functions is relatively small, i.e. from 1 to 35000, compared to the common values for N, i.e. 10^6 or more. The approximations are computed once for each combination of values for D and L. We can subsequently reuse these functions for determining the total execution time, $T'_{cpu}(w)$ or $T'_{gpu}(w)$ for any value of N. It is worth mentioning that in the case of the CPU, for D/L/N = $6/12/5 \times 10^5$, the optimal performance is reached for a grain size of 4096. At the opposite end, a grain size of 1 makes the execution up to 6 times slower. The optimal grain size changes with the input parameters, i.e. for D/L/N = $10/10/5 \times 10^5$, it is 1024. Now it is trivial to discover the optimal grain size, g, that minimizes $T'_{cpu}(w)$. Note that without our optimization we would have to search the grain size that minimizes the execution time for each tuple (D, L, N) we get as input. This means that for every value of the grain size considered in the search we interpolate at a potentially large set of points (e.g. 3×10^6) which can be very time-consuming for a large D or L.

5.2 Static Load Balancing

Static workload balancing eliminates the problems of dynamic workload balancing. What we follow now is to decompose the workload in two partitions. The partitions have sizes proportional to the computational speeds of the CPU and the GPU, or inverse proportional to the execution times. As explained above, the inputs of sparse grid interpolation have a great impact on performance. Hence, they cannot be ignored when determining the speed of the processors.

It is easier to present our approach for static balancing if we consider the execution time functions on the CPU and the GPU as functions of 3 parameters: D, L, and N. To simplify the notations, let us consider that D and L are fixed. We thus have the functions $T_{cpu}(w)$ and $T_{gpu}(w)$ that approximate the execution times on the CPU and the GPU, and take as parameter the number of interpolations. Fig. 4 depicts these 2 functions for various values for the inputs.

Statically solving the workload balancing problem for a given N means finding the value f of w that minimizes $max(T_{cpu}(w), T_{gpu}(N - w)$. If we approximate T_{cpu} and T_{gpu} with 2 linear functions T''_{cpu} and T''_{gpu} (Fig. 4) then it is trivial to find f in $\mathcal{O}(1)$ since it is equivalent to intersecting 2 linear functions. Even for more advanced approximations, determining f can be achieved in linear time.

To achieve efficient static balancing, our goal is to determine the execution time functions as accurate and fast as possible. Consequently, the problem must be reduced to a size that allows us to build the approximations in a minimum amount of time. To obtain accuracy, the reduced problem has to provide results that expose a global behavior, i.e. they are applicable to larger problems. Note that the search for f must be performed for each pair (D, L) so we can consider it as the nest of two loops iterating over a range for D and a range for L.

On the CPU, approximating the execution time is straight-forward since the maximum speed is reached for a relatively small number of interpolation points, leading to the linear behavior visible in Fig. 4 (left). In contrast, on the GPU the large number of active threads (approximately 23040) creates the stepping effect from Fig. 4 (right). For an accurate approximation of the execution time on the GPU, we consider two points: the execution time for N = 1 and the execution time for N = maximum number of active threads + 1. Both measurements include the initial transfer of the compressed data. This ensures a proper approximation that covers the main characteristics of the GPU: the overhead generated by transferring the compressed data to the GPU over PCIe, the high throughput character of the GPU expressed through a large number of SIMD units, and multithreading on the GPU that can improve the performance.

6 Evaluation

We now describe our experimental setup and results. The tested hardware is:

- a system containing a Quad-core Intel Nehalem i7-920 (2.67 GHz) and an Nvidia GTX480 (1.4 GHz, 15 cores, 32-lane SIMD)
- a system with 8 Intel Xeon L5630 cores (2.13 GHz) arranged in two sockets and an Nvidia Tesla X2500 (1.15 GHz, 14 cores, 32-lane SIMD).

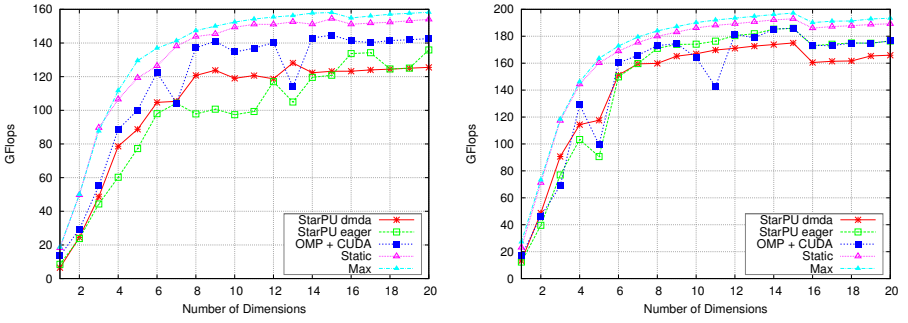


Fig. 5. Left: GFlops rate on 2 × Intel Xeon Quad-core + Nvidia Tesla x2050. Right: GFlops rate on Nehalem Quad-core + Nvidia GTX480

Our application is compiled using gcc 4.4 and nvcc 3.2.

Regarding the problem size, in each run of our application we perform 3×10^6 interpolations. The number of dimensions, D is in the range from 1 to 20 while the refinement level, L is 6. The dynamic approach is implemented using a combination of StarPU and CUDA and a mix of OpenMP and CUDA. From StarPU we only use the fastest 2 schedulers for our application: eager and dmda.

The optimal grain size was determined for each value of D both through brute search and through our optimized search described in Sec. 5. Both searches returned similar optimal grain sizes decreasing from 44000 to 7500, for D between 1 and 20 respectively. These numbers follow to some extent the maximum number of active threads (on the GPU) for D in the range from 1 to 20. Remember that increasing D causes the decrease of the number of active threads. It is worth mentioning that setting the optimal grain size as the maximum number of active threads cannot provide performance portability. On our heterogeneous systems, interpolating on GPU is between 4 and 8 times faster than interpolating on CPU. It is likely that on other systems where the CPU is faster than the GPU, the optimal grain size does not match the maximum number of active threads but instead has a value that permits for the best exploitation of CPU caches.

We can see in Fig. 5 that static workload balancing delivers better performance than the dynamic approach. It is up to 25% times faster than the dynamic version. We attribute this difference to the latency overhead resulting from invoking a significantly larger number of copies to / from the GPU and a larger number of launches of our CUDA program. The amount of transferred data is the same in both approaches but in the static one, only one transfer is necessary.

The max line is a plot of the sum between the GFlops rates of the CPU, $GFlops_{cpu}$, and of the GPU, $GFlops_{gpu}$. To obtain $GFlops_{cpu}$ we run ony the CPU version of interpolation. Similarly, we compute $GFlops_{gpu}$ by executing only on the GPU. Note that the line for the static approach is very close to the max line. More exactly, our static approach reaches up to 98% efficiency defined as: $E = GFlops_{static}/(GFlops_{cpu} + GFlops_{gpu})$. This confirms that the linear approximations from the static approach are sufficiently accurate.

7 Conclusion

In this paper we addressed the workload balancing problem on systems with CPUs and GPUs in the context of sparse grid interpolation. We described static and dynamic task based approaches for load balancing. We showed that input parameters strongly influence the performance of interpolation and the optimal values for load balancing parameters. One such parameter of the dynamic approach is the grain size that can severely reduce the performance on heterogeneous systems. We presented a performance model that helps us to determine the optimal value of the grain size in an acceptable amount of time. Our static approach also enables us to cope with the grain size problem and is built around linear approximations of the execution times on CPU and GPU as functions of workload. Results show that for interpolation, static balancing delivers up to 25% more performance than the dynamic task based strategy.

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

References

1. Garland, M., Kirk, D.B.: Understanding Throughput-oriented Architectures. Commun. ACM 53, 58–66 (2010)
2. OpenMP Application Programming Interface (2008)
3. NVIDIA. CUDA Programming Guide 4.0 (2011)
4. Khronos. The OpenCL Specification 1.1 (2010)
5. Bungartz, H.-J., Griebel, M.: Sparse Grids. Acta Numerica 13(-1), 147–269 (2004)
6. Murarasu, A.F., Weidendorfer, J., Buse, G., Butnaru, D., Pflüger, D.: Compact Data Structure and Scalable Algorithms for the Sparse Grid Technique. In: PPOPP, pp. 25–34 (2011)
7. MAGMA, Matrix Algebra on GPU and Multicore Architectures, `http://icl.cs.utk.edu/magma/index.html`
8. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
9. Osman, A., Ammar, H.: Dynamic Load Balancing Strategies for Parallel Computers. In: ISPDC, Romania (July 2002)
10. Butnaru, D., Pflüger, D., Bungartz, H.-J.: Towards High-Dimensional Computational Steering of Precomputed Simulation Data using Sparse Grids. Procedia CS 4, 56–65 (2011)
11. Intel. Intel Advanced Vector Extensions Programming Reference (2011)