### Title

Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes




### Journal

IEEE Micro, 35(4)

### ISSN

0272-1732



### Authors

Langguth, Johannes
Sourouri, Mohammed
Lines, Glenn Terje
et al.



### Publication Date

2015

### DOI

10.1109/mm.2015.70

Peer reviewed

# Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes

Multicore CPUs can be combined with GPUs to perform computations over 3D unstructured meshes on heterogeneous CPU-GPU clusters. The authors explain how to unlock the CPUs' computing power without slowing down other tasks related to data movement. By solving the representative diffusion equation using the cell-centered finite volume method, the authors demonstrate that combining the computing capacity of CPUs and GPUs delivers a performance advantage over the GPU-only approach.

**Johannes Langguth**
**Mohammed Sourouri**
**Glenn Terje Lines**
Simula Research Laboratory

**Scott B. Baden**
University of California, San Diego

**Xing Cai**
Simula Research Laboratory

General-purpose GPUs as hardware accelerators have entered the high-performance computing landscape in the past few years. In a GPU-enhanced cluster, a computing node typically has significantly higher performance than a node of a homogeneous CPU-based cluster, thus resulting in a denser packing of the heterogeneous clusters. This allows significant cost and power savings due to smaller interconnects.

Owing to the large difference in theoretical floating-point capability between GPUs and CPUs, in computing-bound applications there is little incentive to include CPUs for sharing the computational work on a GPU-enhanced cluster, because this invariably makes the implementation more complex. However, for computations whose performance is limited by data traffic, rather than floating-point operations, CPUs' computing capability should not be overlooked, because the GPU-CPU difference in memory bandwidth is considerably smaller. Four important questions arise regarding heterogeneous GPU-CPU computation:

- How much computational work should be assigned to the CPUs?
- If there are different types of operations, which should be placed on the GPU?
- How should the different tasks on the CPU side be programmed?
- How much performance improvement can we realistically expect from heterogeneous CPU-GPU computing?

Extending our earlier work on single GPU-enhanced computing nodes,[1] in this article, we will shed some light on these questions for heterogeneous clusters. Of course, it is impossible to answer these questions in general. The answers depend on the specific



Published by the IEEE Computer Society

computational problem to be solved and the heterogeneous system's hardware configuration. By choosing a representative case of solving the diffusion equation using the cell-centered finite volume method over unstructured meshes, we aim to provide advice on good programming practices, as well as some important OpenMP and CUDA programming details that carry over to many similar problems.

Moreover, we will also discuss the important issue of CPU-GPU workload partitioning, and we will report performance measurements on up to 128 GPU-enhanced computing nodes, demonstrating the actual performance benefit that heterogeneous CPU-GPU computing offers. For structured meshes, heterogeneous CPU-GPU computation has been studied in several publications.[2–5] However, the unstructured nature of our problem poses significant additional challenges with respect to partitioning, communication, and load balancing.

## Solving diffusion equations with finite volumes

As a representative computational problem, we use the following diffusion equation, which describes a common natural phenomenon:

$$\frac{\partial u(x,t)}{\partial t} = \mathrm{div}\left(\overrightarrow{K}(x)\mathrm{grad}\ u\right), \qquad (1)$$

where $u(x,t)$ is typically some concentration modeled as a function of space and time, and $\overrightarrow{K}(x)$ denotes a spatially varying tensor field that, together with the concentration gradient, determines the speed and direction in which high concentration spreads toward low concentration. Because it is a basic building block of many sophisticated mathematical models, the diffusion equation (Equation 1) is an important research topic for fast numerical solvers and efficient software implementations.

In this article, we consider a finite-volume approach for numerically solving Equation 1 in 3D, using an unstructured tetrahedral mesh. Without going into the mathematical details, suffice it to say that we can represent the actual computation per time step using a matrix-vector multiply:

$$\mathbf{u}^l = Z\mathbf{u}^{l-1}, \qquad (2)$$

where the superscript $l$ denotes the time level and the $\mathbf{u}$ vector contains the numerical approximations at the center of all tetrahedra. Matrix $Z$ is sparse and has, in addition to a nonzero main diagonal, up to 16 nonzero values per row. These 16 off-diagonal nonzeros correspond to each tetrahedron's four immediate neighbors and 12 second-level neighbors.

Throughout this article, we assume that the main diagonal of $Z$ will be stored in a separate 1D array, $D$, whereas the off-diagonal entries are stored in a padded, dense $N \times 16$ array, $A$, with $N$ being the total number of tetrahedra in the mesh. In an unstructured tetrahedral mesh, the column positions of these off-diagonal entries do not follow any easily predictable pattern. Thus, they must be stored separately in $I$, an $N \times 16$ array of integer index values.

The following code segment gives a plain implementation of Equation 2:

```
for (i=0; i<N; i++){
  double value = D[i]*u_old[i];
  for (j=0; j<16; j++)
    value+= A[i,j]*u_old[ I[i,j]];
  u_new[i] = value;
}
```

To calculate `u_new` for each tetrahedron, 33 floating-point operations (17 multiplications and 16 additions) are needed in every time step. At least 208 bytes per tetrahedron must be read from memory (that is, 128 bytes for the 16 `A[i,j]` values, 64 bytes for the 16 `I[i,j]` values, 8 bytes for `D[i]`, and 8 bytes for `u_old[i]`). In case cache data reuse is not perfect, more data must be loaded, depending on the access pattern of the off-diagonal `u_old` values. Moreover, 8 bytes (due to `u_new[i]`) are written to memory per tetrahedron. Thus, computational intensity is at most 33 flops per 216 bytes, and thus, 0.15. This is far lower than the ratio between flops and memory bandwidth in Gbytes per second of modern processors, which starts at 2 and can be higher than 5 for GPUs. Therefore, the theoretical upper limit of the performance for this computation on almost any computing device is

$$P = \frac{33\ \text{flops} \times \text{memory bandwidth}}{216\ \text{bytes}}. \qquad (3)$$

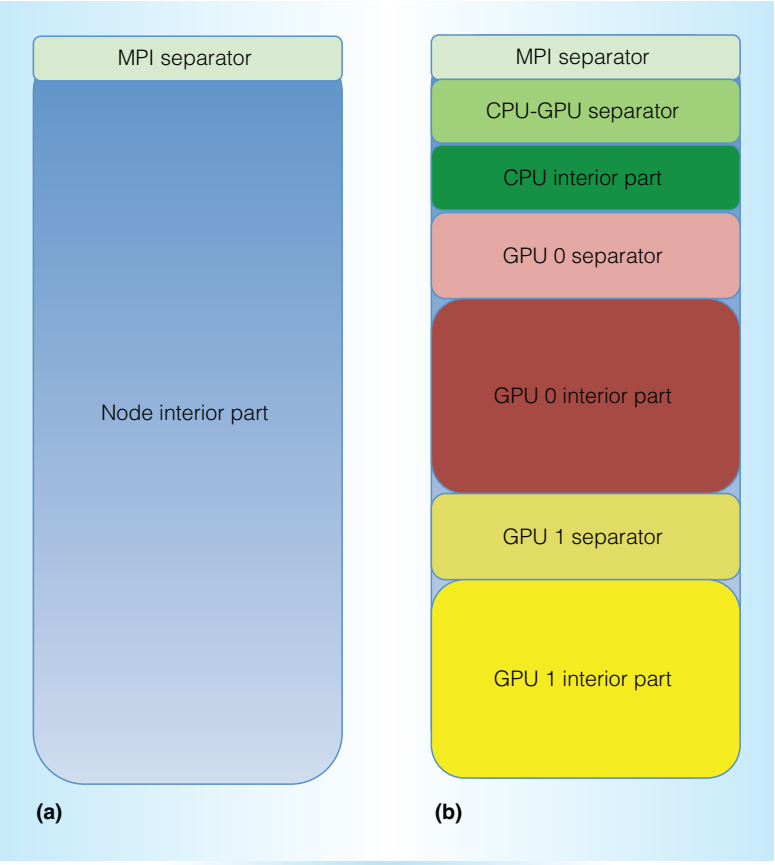


Figure 1. Hierarchical partitioning. (a) Workload per computing node after the initial symmetric partitioning. The cells have been permuted such that the message passing interface (MPI) separator forms a contiguous block. (b) Division of the workload after the intranode partitioning and appropriate permutations.

## Hierarchical mesh partitioning

Given a global tetrahedral mesh, the first step in parallelization is a standard $k$-way partitioning, where $k$ is the number of available computing nodes. Then, for each node $n$, the subdomain matrix $Z_n$ can be independently computed corresponding to the tetrahedra that the mesh partitioner has assigned to $n$. Because of internode connections, the rows of $Z_n$ that rely on values of tetrahedra assigned to neighboring nodes constitute a separator, whereas the remaining rows of $Z_n$ constitute the interior part. Tetrahedra from other nodes that have neighbors on node $n$ are replicated as ghost cells on $n$. During each time step of the computation, using asynchronous message passing interface (MPI) communications, node $n$ receives updated values for these ghost cells from neighboring nodes and sends updated values of its separator cells in return. The elements to be sent to a specific node must be packed into a contiguous buffer during every round. The ghost cells are organized such that their values can be received in a contiguous fashion.

In order to use heterogeneous computing, a second tier of partitioning is required to split the CPU part from the accelerator part on each node. Here, an asymmetric partitioning is needed, because the GPUs will generally receive a higher workload than the CPUs. On each node, we generate one subpartition per GPU, plus one CPU subpartition. In the case of multiple CPU sockets, the CPUs can work on a shared subpartition using OpenMP, although care must be taken to obtain their full performance.[6] Although our test systems' nodes have at most two GPUs and CPUs, our code can deal with almost any configuration as long as enough CPU threads are available.

Thus, the global mesh is broken into $k$ parts, each of which is subdivided into the MPI separator, a CPU-GPU separator, and a CPU interior part, and for each GPU, a GPU-CPU separator and a GPU interior part (see Figure 1). All the resulting separators are packed together to allow contiguous access for computation and communication. There is no explicit GPU-GPU separator, because this communication is performed by transferring data via the CPU.

Although several techniques exist for direct communication between accelerators (for example, GPUdirect[7]), we did not use them because they tend to be hardware specific and thus offer little portability.

## Heterogeneous implementation

The CPU part of a computing node, which can comprise several physical sockets working on shared memory, handles all communication between the GPUs, as well as the MPI communication with other nodes using a single MPI process. This technique's advantage is that the entire complexity of running the heterogeneous computation is encapsulated in the intranode code. Thus, existing internode communication schemes can be reused when transforming conventional codes into heterogeneous implementations. In our case, a

simple set of `MPI_Isend` and `MPI_Irecv` instructions handle internode communication. In addition to its simplicity, this approach has the advantage of keeping the number of MPI ranks—and thus, the total size of the separators—low, which ensures that communication is unlikely to become a bottleneck.

Finally, to ensure good cache data reuse for the off-diagonal `u_old` values, we use the partitioner for a third time to reorder the tetrahedra in the interior computation parts. The goal is to create blocks of tetrahedra that have as many neighbors as possible within each block, and thus as few neighbors as possible outside the block. Doing so regularizes accesses to `u_old[i]`, which has a dramatic effect on performance.[6,8] We obtained good performance for a block size of 512 tetrahedra on the CPU and 64 on the GPU. Note that only the tetrahedra in the interior computation part are reordered in this way. Reordering the tetrahedra in the separators is not worthwhile, because they invariably access neighbors outside the current device—and thus, outside their block.

Our GPU kernel processes elements of $A$ in a column-major ordering, as suggested in Vázquez et al.[9] This means that for every thread block of size $b$, every $b$ contiguous rows are turned into a $b \times 16$ submatrix and transposed. This allows coalesced accesses to values of $A$ and $I$—that is, threads in a thread block access the elements in a contiguous manner, thus attaining full memory bandwidth. We found that a thread block size $b = 128$ yielded the best performance, because smaller sizes limit the device occupancy. Tetrahedra beyond the last block of size $b$ are computed using a row-major kernel. Because of their small number, they have a negligible effect on performance.

Our heterogeneous code's core is the assignment of different tasks to different hardware CPU threads. In our implementation, we do this by directly assigning a type to a thread on the basis of its OpenMP thread number. We use one control thread per accelerator. Each such control thread launches the separator computation on its accelerator, then copies the result asynchronously to the CPU, and starts the computation of the interior part. Meanwhile, all the remaining threads work on the MPI separator elements. When this is done, a single thread diverges and communicates the `u_new` values belonging to the MPI separator to the neighboring nodes via MPI, while receiving corresponding values in return. In our experiments, using more than one MPI communication thread did not pay off.

The remaining threads then compute the CPU-GPU separator, and upon completion, one copy thread per accelerator diverges to start copying the result to its accelerator, while the remaining threads compute the interior CPU part, which means that they are pure computational threads. Each copy thread also transfers `u_new` values belonging to the separators from other accelerators to its own accelerator once they have been transferred to the CPU memory. Because these transfers are asynchronous, the copy thread can then rejoin the computational threads working on the interior CPU part. When all these tasks have been executed, the threads are gathered at a barrier. The array pointers of `u_old` and `u_new` are then swapped on all devices, and a new time step begins.

Note that we only use physical cores to run the threads. Hyperthreading and similar techniques could make the threads less responsive, and can thereby reduce performance. Thus, for a given number of accelerators, an equal number of control and copy threads must be available in addition to the computational threads. If too few computational threads remain, the CPU performance will be low, which invalidates the entire approach. As a rule of thumb, the total number of cores should be at least four times the number of accelerators. Figure 2 shows an overview of the threads for a typical test node.

In addition to this, the GPU control threads (threads 14 and 15 in Figure 2) and the GPU communication threads (threads 1 and 2 in Figure 2) use multiple CUDA streams to overlap communication and computation on the GPU. Thanks to its two copy engines, a modern GPU such as the K20 can send its separator while simultaneously receiving the CPU-GPU separator from its copy thread, as Figure 2 shows. An example of streams that overlap communication with computation can be found in Figure 3. The figure is derived from the output

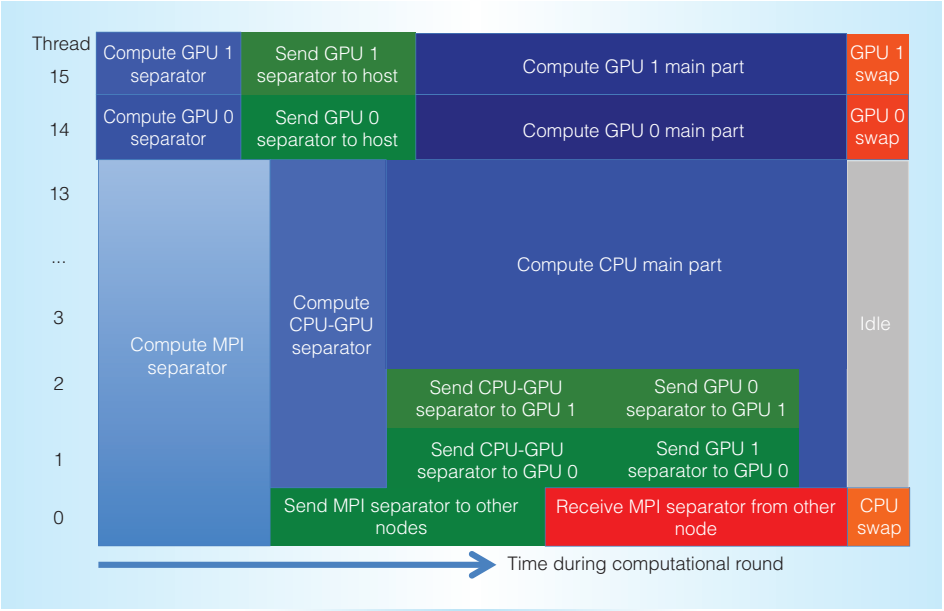


Figure 2. OpenMP task assignment. Example of the task-parallel thread assignment using 16 cores and two GPUs. Threads 14 and 15 serve as control threads for the GPUs, and threads 1 and 2 are their copy thread. Threads 3 through 13 are computational threads and perform only computation, and thread 0 is the MPI communication thread.

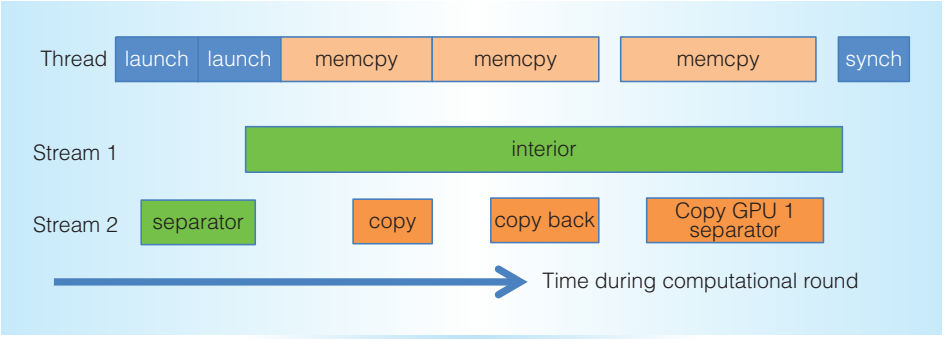


Figure 3. CUDA streams. Using multiple streams to overlap communication and computation. Instructions issued in the CPU thread are assigned to different GPU streams to overlap communication and computation on the GPU.

of the *nvprof* GPU profiler. Some details have been modified for visibility—for example, the interior tetrahedra take far longer to compute than all other operations combined, but it has been shortened here. The interior computation cannot be overlapped with the separator computation because both use the same computing resources, but it does overlap with communication. While the kernel launches are relatively fast compared with the kernel running time, initializing CPU-GPU data transfer incurs a significant overhead in the calling thread, even though the transfer itself is fast. We also observe synchronization gaps (that is, periods in which no computation occurs). These synchronization costs represent a significant challenge for achieving high performance.

## Experimental setup

All experimental instances are derived from a 3D mesh of a healthy male human cardiac geometry acquired by MRI. We employ

*tetgen* to generate the initial global mesh.[10] For our experiments, we set a target resolution via a maximum volume constraint per tetrahedron of $2.8 \times 10^{-6}$, thereby generating more than 115 million tetrahedra. This test instance is large enough for a moderate number of computing nodes. However, we need at least five GPUs to store the partitioned data in device memory at this instance size. To obtain measurements on smaller node counts, we created a second instance of 6.8 million tetrahedra, which can be run using a single GPU. We run all experiments using fewer than 8 GPUs using this smaller instance instead.

We then use the PaToH[11,12] and Kaffpa[13] partitioning software to generate the initial $k$-way partitioning of the global mesh. Kaffpa generally takes less time to partition than PaToH and produces better quality (if we use the high-quality setting). Because Kaffpa can only generate symmetric partitions, we use PaToH for the intranode partitioning and then use Kaffpa again for the reordering.

Finally, to maintain generality, we do not exploit the effects of fitting the entire CPU workload in the L3 cache. Small CPU workloads can lead to high CPU performance when all required data fits in cache.[1] Thus, in the case of an extremely small CPU workload, it is worthwhile to expand it up to the maximum cacheable size, thereby speeding up the overall computation. Although we do not use this, this effect guarantees that, in the future, the CPU can remain useful for memory-bound computations even if many fast accelerators are available. Another way to benefit from large CPU caches might be to explicitly load the MPI separators into cache to compute them quickly at the start of each round.

We use two heterogeneous machines with slightly different characteristics as test hardware systems, which lead to different ratios between CPU and GPU workload in these systems. This is important for benchmarking our heterogeneous code.

As our primary test system, we use the GPU part of TACC's Stampede. It is primarily a Xeon Phi machine, but we use its GPU partition for our experiments. Each of its 128 GPU nodes possesses a single Nvidia K20 GPU and two powerful Intel Xeon E5-2680 processors with eight cores each, which gives it a strong CPU-to-GPU performance ratio.

As the secondary machine, we use the University of Cambridge's Wilkes system. We use up to 64 nodes on Wilkes, and each node has two CPUs and two Nvidia K20 GPUs, only one of which can be accessed at full PCI Express (PCIe) bus speed from a given CPU. Thus, each CPU has one preferred GPU, and the second GPU is accessed through the other CPU on the node. The CPUs are Intel Xeon E5-2630v2 (that is, Ivy Bridge processors), which are similar to the Sandy Bridge processors used in Stampede. However, these CPUs have only six cores each and lower attainable memory bandwidth, which reduces the system's CPU-to-GPU performance ratio. On both machines, we use the Intel icc compiler 13.1.0, Intel MPI 4.1.3.049, and CUDA 6.0. We deactivate hyperthreading in all instances, and use one OpenMP thread per core. We set the OpenMP thread affinity to "scatter." We use up to 64 nodes on Wilkes.

## Experimental results

Using the test setting described earlier, we tested the performance of the individual computing devices to assess load balancing and upper limits on performance. We then tested scaling performance using only CPUs and only GPUs. Our final experiment measured the heterogeneous performance and compared it to the GPU-only values, thereby evaluating the success of our heterogeneous scheme.

### Single-device computation performance test

A crucial ingredient of our heterogeneous implementation is the static workload ratio, which we obtain using performance predictions that are based on the computing devices' memory bandwidth. For any device, we obtain its workload ratio by dividing its predicted performance $P$ by the sum of the predictions for all devices. For convenience, we denote the total GPU workload ratio as $r$, which means the CPU workload ratio will be $1 - r$ and each individual GPU will have a workload of $r$ divided by the number of GPUs.

Now, given the peak memory bandwidth provided by the vendors and the fact that the maximum flop-to-byte ratio is 0.157, we obtain $P_{\text{peak}}$, that is, the predicted performance

**Table 1. Computational performance estimates ($P_{peak}$ and $P_{stream}$) and measurements ($P_{real}$) of a single device in each of the test systems (in gigaflops). The $r$ values denote workload partitioning ratios computed on that basis. Unlike the GPU peak bandwidth, the GPU stream bandwidth is based on activated error-correcting code.**

| Bandwidth (Gbytes per second) | Stampede | Wilkes |
|---|---|---|
| CPU peak bandwidth | 102.4 | 102.4 |
| CPU stream bandwidth | 77.8 | 72.9 |
| GPU peak bandwidth | 208 | 2 × 208 |
| GPU stream bandwidth | 151.1 | 2 × 151.1 |
| **Performance (Gflops)** | **Stampede** | **Wilkes** |
| CPU $P_{peak}$ | 16.11 | 16.11 |
| CPU $P_{stream}$ | 12.24 | 11.47 |
| CPU $P_{real}$ | 11.46 | 8.78 |
| GPU $P_{peak}$ | 32.74 | 2 × 32.74 |
| GPU $P_{stream}$ | 23.78 | 2 × 23.78 |
| GPU $P_{real}$ | 21.46 | 2 × 21.46 |
| **Workload divisions** | **Stampede** | **Wilkes** |
| $r_{peak}$ | 0.67 | 0.80 |
| $r_{stream}$ | 0.66 | 0.80 |
| $r_{opt}$ | 0.65 | 0.83 |

based on these values, and thus the appropriate workload ratio $r_{peak}$. Table 1 shows the results. We compare this to the actual measured performance $P_{real}$ and the resulting optimal workload ratio $r_{opt}$. We can use $P_{real}$ to obtain an upper limit on the heterogeneous code's performance by multiplying the corresponding $P_{real}$ values with the number of computing devices used.

The discrepancy between $P_{peak}$ and $P_{real}$ is significant, which implies that $P_{peak}$ is not a good performance prediction. We improve it by using $P_{stream}$, which is the performance estimate based on bandwidth measured using the Stream benchmark.[14] Table 1 clearly shows that $P_{stream}$ is a much better prediction for $P_{real}$, and $r_{stream}$ is closer to $r_{opt}$.

Interestingly, the difference between the workload ratios is small in most cases. However, overestimating CPU performance by even a small amount has a comparatively large impact on the overall performance when the CPU contribution is small.[1] For example, on Wilkes, the fact that both $P_{peak}$ and $P_{stream}$ overestimate the CPU performance leads to CPU workloads that are 17 percent higher than optimal (that is, from 0.17 to 0.2). This could in turn lead to roughly 17 percent higher execution time and thus 15 percent lower performance. Thus, we conclude that benchmarking the actual performance $P_{real}$ to obtain $r_{opt}$ can pay off, and we use it in this study, but it might not be worthwhile in practice. In general, we recommend reducing the CPU workload a bit because erring in the direction of high CPU workloads is much more costly than vice versa.

### Homogeneous node scaling experiment

In the previous experiments, we obtained a theoretical upper bound on performance. Now we bound it from below by running the full communication and computation on the test systems, but we use only CPUs or GPUs, thereby establishing the maximum performance attainable without using heterogeneous computing. This is necessary to assess the performance gain—and, thus, the potential payoff in using heterogeneous CPU-GPU computation. Figure 4 shows the attained performance on both Stampede and Wilkes.

These results include MPI communication, and are thus significantly lower per node than the $P_{real}$ values from Table 1 would indicate. Summing up the CPU and GPU values gives us an estimate for the performance upper limit we can expect from heterogeneous computing. Interestingly, despite having the same GPUs and using only one GPU per node on both machines, we observe noticeably lower GPU performance on Wilkes.

### Heterogeneous node scaling experiment

Figure 5 shows the results for our heterogeneous implementation and compares the attained performance to using only GPUs, and to an instance of the heterogeneous code where all communication is disabled. On Stampede, the difference between the heterogeneous and pure GPU results is quite pronounced, which validates our technique's usefulness. Furthermore, the communication-free performance is only slightly higher, which indicates that communication is largely overlapped with computation. The speedup for 128 nodes is 98.7.

For Wilkes, we obtained the GPU-only value by running the pure GPU code with

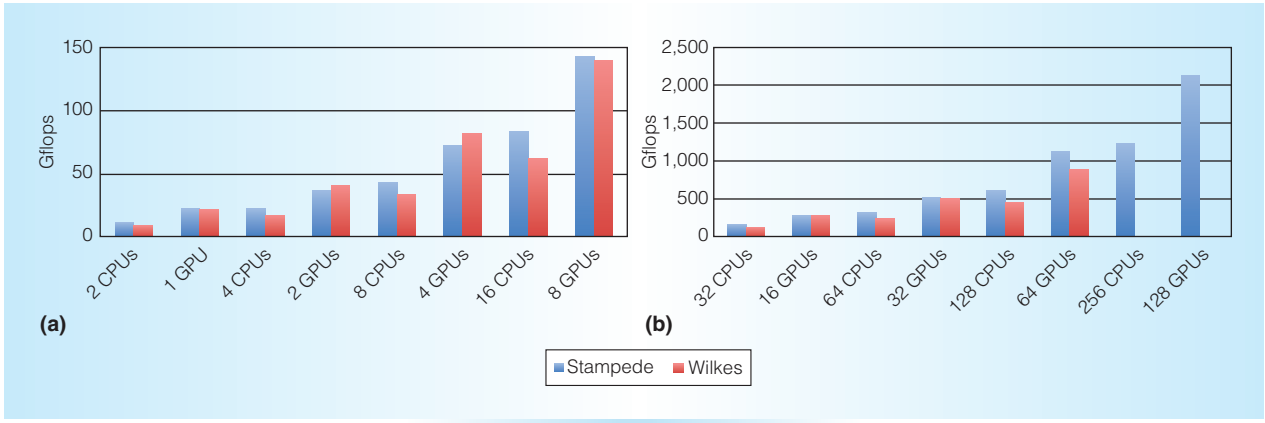


Figure 4. Homogeneous performance. (a) Small instances. (b) Large instances. Note that CPUs refers to the number of sockets, not cores.

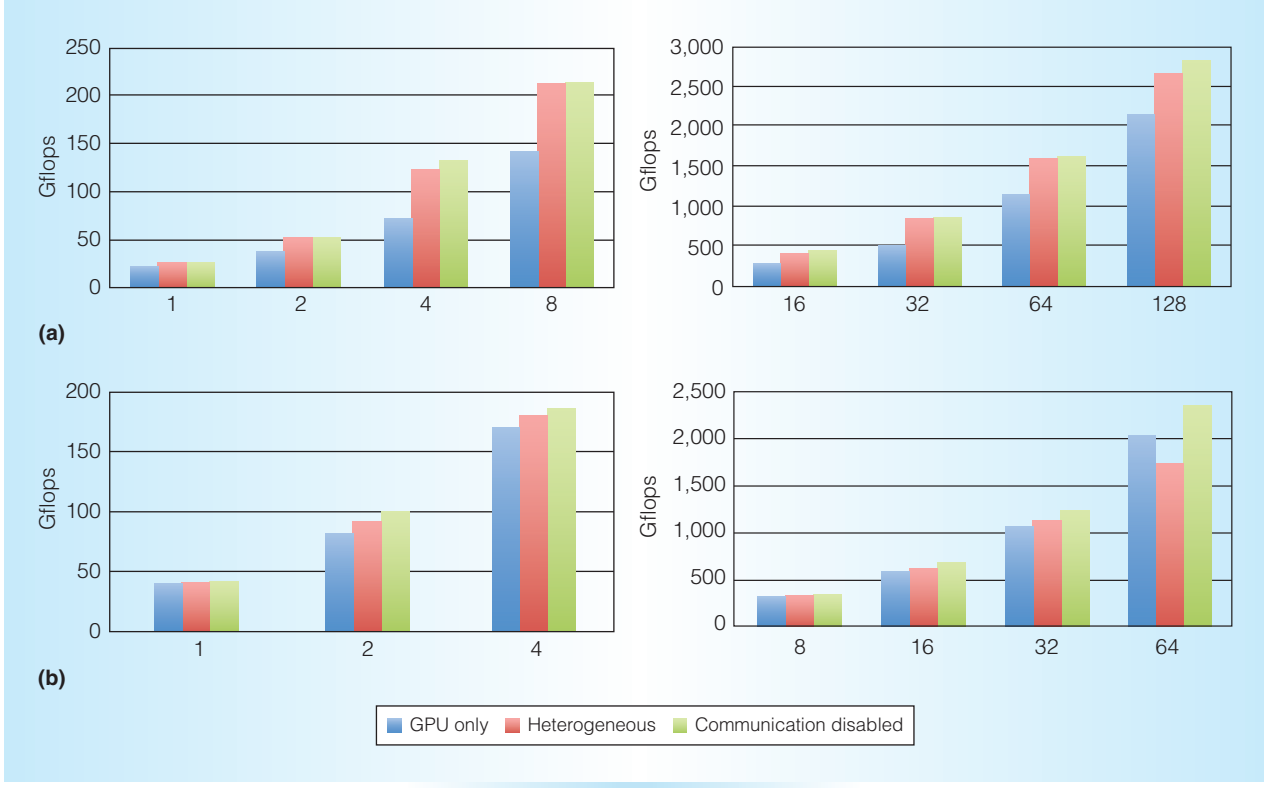


Figure 5. Heterogeneous performance: simulation results for (a) Stampede and (b) Wilkes. Results are shown in relation to GPU-only and communication-free performance using up to 128 GPUs and nodes.

two MPI processes on each node. The process placement is such that it matches each process with its preferred GPU, thus optimizing communication performance. The more complex node layout, along with the fact that the CPUs are weaker on this machine, reduces the heterogeneous code's performance lead. The speedup is 27.7 for 32 nodes and 42.7 for 64 nodes. Also, for 64 nodes, the heterogeneous performance is actually lower than the pure GPU result, while the communication-free performance is significantly higher. This indicates that in this setup, intranode communication is in fact a bottleneck.

We assume that this bottleneck is due to limitations in strong scaling—that is,

workloads per computing device become so small that communication becomes an issue in this case. In addition, the CPUs on Wilkes essentially have nonuniform memory access to the GPUs, which our assignment of control threads does not consider. Furthermore, our one MPI thread communication model is not ideally suited to fully use the two InfiniBand adapter cards per node. This suggests that for complex computing nodes, the scheme must be adapted to obtain full communication performance.

When considering parallel efficiency, which we can derive by multiplying the single-device performance results from Table 1 with the number of devices involved, we find that the value remains quite stable at about 80 percent. However, the value starts to drop when at least 64 GPUs are involved. We assume that this is the general limit for strong scaling at this instance size, because the communication-free results show similar behavior, which indicates that on Stampede, the principal limitations to strong scaling are caused by load-balancing and synchronization issues, rather than communication overhead.

Consequently, future work will focus on improving load balancing to maintain scalability for even larger instances on very large clusters. However, because our heterogeneous parallelization scheme is in no way limited to the example application discussed in this article, our main concern is to make the system available for a wide range of scientific applications on heterogeneous clusters.

Our experiments on Stampede show that the strong scalability is very good when using up to 32 GPUs. At 128 nodes, we still attain 95 percent of the communication-free upper bound. Efficient communication is a concern on the complex nodes of Wilkes, though.

Although the chosen diffusion equation and the explicit finite-volume numerical strategy are simple, the obtained experiences with hierarchical mesh partitioning, CPU-GPU workload division, and OpenMP/CUDA programming readily extend to more advanced real-world applications. One possible direction of future work is to apply our findings to the monodomain model of computational electrocardiology, which comprises the diffusion equation and a set of ordinary differential equations that describe the electrical behavior of cardiac cells.

We have focused on a single application, but the programming techniques we describe in this article are not application specific. Assuming that static load balancing is suitable for the problem, and interior cells significantly outnumber separator cells, the techniques we describe can be used to efficiently incorporate CPU and GPU operations on many kinds of mesh-based computations. MICRO

## Acknowledgments

We thank Julius Guccione at the University of California, San Francisco, for providing segmented surfaces from cMRI images, which we used as our test cases. We thank Filippo Spiga at the University of Cambridge for his support in running experiments on Wilkes. This work used the Wilkes GPU cluster at the University of Cambridge High Performance Computing Service, provided by Dell, Nvidia, and Mellanox, and partly funded by the STFC, with industrial sponsorship from Rolls-Royce and Mitsubishi Heavy Industries. We thank the Texas Advanced Computing Center at UT Austin for providing HPC resources that contributed to our research. Scott Baden dedicates his portion of this work to Ingela Brising (1943–2015).

## References

1. J. Langguth and X. Cai, "Heterogeneous CPU-GPU Computing for the Finite Volume Method on 3D Unstructured Meshes," *Proc. 20th IEEE Int'l Conf. Parallel and Distributed Systems*, 2014, pp. 191–199.
2. A. Humphrey et al., "Radiation Modeling Using the Uintah Heterogeneous CPU/GPU Runtime System," *Proc. 1st Conf. Extreme Science and Eng. Discovery Environment*, 2012, pp. 4:1–4:8.
3. X. Yue, S. Shu, and C. Feng, "UA-AMG Methods for 2-D 1-T Radiation Diffusion Equations and Their CPU-GPU Implementations," *Proc. 21st Int'l Conf. Nuclear Eng.*, 2013; doi:10.1115/ICONE21–16157.
4. M. Wen et al., "High Efficient Sedimentary Basin Simulations on Hybrid CPU-GPU Clusters," *Cluster Computing*, vol. 17, no. 2, 2014, pp. 359–369.

5. T. Shimokawabe et al., "Petascale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer," *Proc. Int'l Conf. High Performance Computing, Networking, Storage, and Analysis*, 2011, pp. 3:1–3:11.
6. J. Langguth et al., "Parallel Performance Modeling of Irregular Applications in Cell-Centered Finite Volume Methods over Unstructured Tetrahedral Meshes," *J. Parallel and Distributed Computing*, vol. 76, 2015, pp. 120–131.
7. *Nvidia GPUDirect Technology Accelerating GPU-based Systems*, tech. report, Mellanox Technologies, 2010.
8. J. Langguth et al., "On the GPU Performance of Cell-Centered Finite Volume Method over Unstructured Tetrahedral Meshes," *Proc. 3rd Workshop Irregular Applications: Architectures, and Algorithms*, 2013, pp. 7:1–7:8.
9. F. Vázquez et al., "Improving the Performance of the Sparse Matrix Vector Product with GPUs," *Proc. IEEE 10th Int'l Conf. Computer and Information Technology*, 2010, pp. 1146–1151.
10. H. Si, *TetGen: A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator*, 2007; http://wias-berlin.de/software/tetgen.
11. U.V. Çatalyürek and C. Aykanat, "A Hypergraph Model for Mapping Repeated Sparse Matrix-Vector Product Computations onto Multicomputers," *Proc. Int'l Conf. High Performance Computing*, 1995.
12. U.V. Çatalyürek and C. Aykanat, "A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices," *Proc. 15th Int'l Parallel and Distributed Processing Symp.*, 2001, p. 118.
13. P. Sanders and C. Schulz, "Think Locally, Act Globally: Highly Balanced Graph Partitioning," *Proc. 12th Int'l Symp. Experimental Algorithms*, LNCS 7933, 2013, pp. 164–175.
14. J.D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995, pp. 19–25.

**Johannes Langguth** is a postdoctoral fellow at Simula Research Laboratory. His research interests include computer architecture, parallel algorithms, combinatorial optimization, and high-performance scientific computing on multicore CPUs and GPUs. Langguth has a PhD in computer science from the University of Bergen. Contact him at langguth@simula.no.

**Mohammed Sourouri** is a PhD student at Simula Research Laboratory. He is also a member of the Scientific Computation Group at the University of California, San Diego. His research focuses on parallel computing and domain-specific source-to-source translators. Sourouri has an MSc in computer science from the University of Oslo. Contact him at mohamso@simula.no.

**Glenn Terje Lines** is a senior research scientist at Simula Research Laboratory. His research interests include numerical methods for partial differential equations and computer simulation of cardiac electrophysiology. Lines has a PhD in computer science from the University of Oslo. Contact him at glennli@simula.no.

**Scott B. Baden** is professor in the Department of Computer Science and Engineering at the University of California, San Diego. His research focuses on domain-specific translation, performance programming, and adaptive and irregular applications. Baden has a PhD in computer science from the University of California, Berkeley. He is a senior member of IEEE, a member of SIAM, and a senior fellow at the San Diego Supercomputer Center. Contact him at baden@eng.ucsd.edu.

**Xing Cai** is the head of the HPC group at Simula Research Laboratory and a professor of computer science at the University of Oslo. His research interests include parallel programming and high-performance scientific computing on multicore CPUs and GPUs, numerical methods for solving partial differential equations (PDEs) and generic PDE software. Cai has a PhD in scientific computing from the University of Oslo. Contact him at xingca@simula.no.