ICCM2014
28-30th July, Cambridge, England

Effect of GPU Communication-Hiding for SpMV Using OpenACC

***Olav Aanes Fagerlund[1], Takeshi Kitayama[2,3], Gaku Hashimoto[2] and Hiroshi Okuda[2]**

[1] Department of Systems Innovation, School of Engineering, The University of Tokyo,
7-3-1 Hongo Bunkyo-ku, Tokyo 113-8656, Japan.
[2] Graduate School of Frontier Sciences, The University of Tokyo, 5-1-5 Kashiwanoha, Kashiwa,
Chiba 277-8563, Japan.
[3] Japan Science and Technology Agency, CREST, 7, Gobancho, Chiyoda-ku,
Tokyo 102-0076, Japan.

*Corresponding author: olav@multi.k.u-tokyo.ac.jp

Abstract

In the finite element method simulation we often deal with large sparse matrices. Sparse matrix-vector multiplication (SpMV) is of high importance for iterative solvers. During the solver stage, most of the time is in fact spent in the SpMV routine. The SpMV routine is highly memory-bound; the processor spends much time waiting for the needed data.
In this study, we discuss overlapping possibilities of SpMV in cases where the sparse matrix data does not fit into the memory of the discrete GPU, by using OpenACC. With GPUs one can take advantage of their relatively high memory bandwidth capabilities. However, data needs to be transferred over the relatively slow PCI express (PCIe) bus. This transfer time can to a certain degree be hidden. We concurrently perform computation on one set of data while another set of data is being transferred. Parameters such as the size of each subdivision being transferred - the number of matrix subdivisions, and the whole matrix size, are adjustable. We generate matrices modeling one, three and six degrees of freedom. It is observed how these parameters affect performance. We analyze the improved performance as a result of communication-hiding with OpenACC, and a profiler is used to provide us with additional insight. This is of direct relevance for a block Krylov solver, for instance a block Cg solver. Here, one can benefit from streaming of data with SpMV and overlap while doing so. Each streamed subdivision is used several times with different vectors.
When using a discrete GPU with an ordinary (non-block) Krylov solver, one has to run SpMV once over the whole matrix (or subdivision) for each solver iteration, so there will be no benefit if the matrix does not fit the GPU memory. This is due to the fact that streaming the matrix over the PCIe bus for each of the solver iterations incurs a too big overhead.
For instance, in the case of three degrees of freedom and modeling 2,097,152 nodes, we observe a just above 40% performance increase by applying communication-hiding in our benchmarking routine. This gives us close to 33 GFLOP/s on the AMD Tahiti GPU architecture, in double precision. When modeling the same amount of nodes with a 'synthetic' six degrees of freedom, up to ~65.7% is observed in increased performance when hiding parts of the data transfer time. This underlines the importance of applying such techniques in simulations, when it is suitable with the algorithmic structure of the problem in relation to the underlying computer architecture.

Keywords: SpMV, OpenACC, GPGPU, communication-hiding, overlapping, FEM, block Krylov

Introduction

Sparse Matrix-Vector multiplication (SpMV) is of high importance in many engineering applications. In iterative solvers it plays an important part, as this is the routine where most of the solver-time is spent. In the Finite Element Method (FEM) very large problems are frequently handled. Direct solvers will consume unmanageable amounts of memory when the problems are large, so a iterative solver is the choice here. The time to solution partly depends on a fast SpMV routine. It is an operation that can be parallelized. However, the main challenge is the memory

bottleneck found in the current day systems. For each computation in need of being performed, a significant amount of data must be read from memory. So, we cannot achieve a performance any close to the peak performance of the processor itself. The memory subsystem, and how well it is utilized, gives the limitation as to what speed can be achieved.
To partly get around the problem several sparse matrix storage formats have been developed over the years. These typically only store the non-zero elements and their locations in the matrix, thus saving storage space and bandwidth usage. The storage format of a sparse matrix can greatly influence the performance achieved, as it will dictate the memory access pattern under execution, which again affects performance according to how well that particular memory access pattern suits the underlying architecture executing the code. Today, GPUs deliver unrivaled memory bandwidth paired with massively parallel processors, so running these operations on GPUs can give a substantial advantage. Due to its importance in engineering simulation there has been extensive research in optimizing the SpMV routine [Bell and Garland (2008)]. In order to parallelize our code and also get a more performance-portable code-base we have parallelized our routines by taking advantage of a modern OpenACC compiler and its associated runtime.
One of the main obstacles that remain when using discrete GPUs is the need to transfer data over the relatively slow PCI express (PCIe) data-bus. The latency and bandwidth of this bus is at least an order of magnitude slower than the internal communication on the GPU card between the GPU chip and GPU memory ('global memory' in OpenCL terms, or 'VRAM' in more traditional terms).
For a 16-lane PCI express connection (one 16x slot) the speeds are as follows:

- PCIe 1.x : 4 GB/s
- PCIe 2.x : 8 GB/s
- PCIe 3.0 : 15.75 GB/s
- PCIe 4.0 : 31.51 GB/s (Not released yet)

Even 'on-GPU-card', where the bandwidth between the global memory and the GPU chip can be about 300 GB/s, we are severely memory bandwidth limited. We can only attain a small percentage of the peak GPU performance as we wait for the data needed for the computation. This illustrates well the problem the PCIe bus adds.
To minimize this effect we implement communication-hiding by performing needed memory transfers over the PCIe bus while a previous set of data is computed upon. The aim is to drastically reduce the overall execution time, effectively raising the performance of the SpMV operation as a whole. This can significantly benefit block Krylov solvers, described in [Saad (2003)], where one do multiplications with the same matrix and different vectors several times versus only one time with one vector per iteration as in standard Cg. The many SpMV operations can be performed while a 'new' subdivision is transferred over the PCIe bus, as they combined are time-consuming enough so that the solver as a whole benefits from the overlapping – in cases where the data simply cannot fit the available GPU memory. For instance, 'Stochastic FEM using the Seed Method', described in [Sato and Okuda (2008)], is one such block solver that could benefit from this - in its GPU-based incarnation.

Implementation of communication-hiding

A cube, consisting of smaller cubes, is modeled. The problem is a classical cantilever problem with a force applied. The cube has a certain amount of nodes, set by the number of nodes per side. The number of nodes and one, three or six degrees of freedom for all nodes are parameters set prior to matrix generation. The matrices generated will mimic stiffness matrices commonly encountered in FEM problems, and are highly sparse. Hexahedra elements are used, i.e. the nodes are in the corners of each cube, and each element has six faces. Note that for this problem, six degrees of freedom is a "synthetic" scenario. However, its results are of interest nevertheless.

We keep our data in the CSR-format, a format that is more deeply explained in [Vuduc (2003)], and from where Fig. 1 is inspired. The amount of GPU memory can serve as a limitation, as the stiffness matrix can be many times larger than the total GPU memory. Therefore, we divide the stiffness matrix up into several equal-sized subdivisions, upon transfer to the GPU over the PCI express bus. This serves two purposes:

1. Each subdivision will fit in GPU memory.
2. Communication-hiding can be performed on subdivisions subsequent to the first subdivision transfer. To enable this efficiently two blocks must fit into memory at any given time, in addition to other vectors of the CSR format needed.

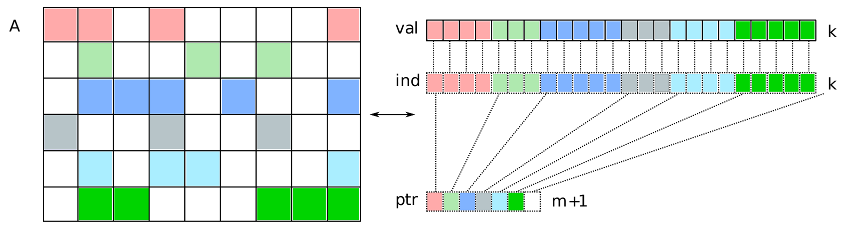


Figure 1. The CSR format layout.

The more nodes we model, the bigger the stiffness matrix becomes. The degrees of freedom decide the amount of blocked non-zero elements per 'node-to-node-connection'. One degree of freedom gives 1 x 1 elements, three degrees of freedom gives 3 x 3 elements, and six degrees of freedom gives 6 x 6 blocked non-zero elements per 'node-to-node-connection'. As we persistently use double precision, each element of the matrix consumes 8 Bytes. The basic structure of the communication-hiding is as follows. Here, for example, we explain the details of how four subdivisions are read on the fly from system main memory:

1. The stiffness matrix is divided into subdivisions. It must be, at a minimum, be divided into enough subdivisions so that two subdivisions can reside in GPU memory concurrently. In our implementation all other vectors of the CSR format are kept in GPU memory without being broken up, as the sizes are of a significantly lesser magnitude than the stiffness matrix.
2. First subdivision is transferred to GPU memory, synchronously.
3. Computation over the first subdivision (2.) is commencing once the transmission of the first subdivision is completed. However, immediately before this takes place the transfer of the second subdivision is initiated asynchronously.
4. Once computation over the first subdivision is completed, the memory space consumed by the first subdivision is de-allocated.
5. It is ensured that the transfer of the second subdivision has completed. Transfer of the third subdivision of data is initiated asynchronously.
6. Computation over the second subdivision of data is commenced and completed.
7. Once the previous step (6.) is completed, the associated second subdivision of data is de-allocated. It is ensured that the transfer of the third subdivision has completed.
8. Transfer of the fourth subdivision of data is initiated asynchronously.
9. Computation over the third subdivision of data is commenced and completed.
10. Once the previous step (9.) is completed, the associated third subdivision of data is de-allocated. It is ensured that the transfer of the forth subdivision has completed.
11. Computation over the forth subdivision of data is commenced and completed. Forth subdivision of data is de-allocated once done. All processing is completed; the complete result vector residing in GPU memory is copied into system main memory.

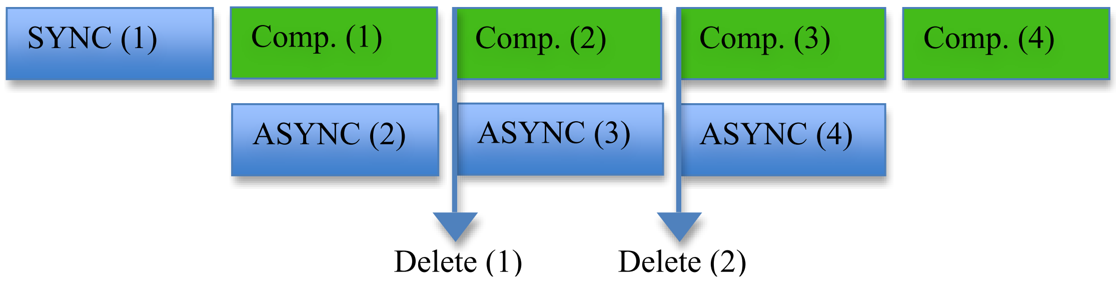


Figure 2. Illustration of the sequence with 4 subdivisions used in communication-hiding.

Fig. 2 shows the sequence explained in detail above, point 1-11. Blue boxes illustrates data transfers, while the green boxes illustrates the computation over the data of the same ID number.

Throughout these stages appropriate ‘book keeping’ is done to ensure correctness, required when the stiffness matrix is broken up and computed upon piece by piece. The sequences perform in a similar pattern when the stiffness matrix is divided into more subdivisions.
It is important to note that our routine will do computations over the data 50 times, for each subdivision transferred over the PCI Express bus. For each of these 50 iterations, all of the values in the vector being multiplied with the matrix are modified, so each iteration yield different result. Thus, the transfer time over the PCI Express bus is partly amortized. With the overlapping applied it is further partly hidden. The behavior mimics that of a block Krylov solver performing SpMV. Of course, if we reduce the amount of iterations, and then consequently the amount of GPU computational work for each block transfer, the communication time over the PCI Express bus will dominate more and influence the performance. This underlines the general importance of reducing the communication, and to perform as much communication-hiding as possible, in heterogeneous computing. With GPU computing we cannot get around the fact that we have to be able to amortize the communication costs over the ‘slow’ PCI Express bus, if the GPU cannot directly access the system main memory with higher bandwidths and lower latencies than the PCI Express bus can supply.

Methodology of benchmarking

Parameters such as the size of each subdivision of data being transferred - the number of matrix subdivisions, and the whole matrix size, are adjustable. We generate matrices modeling one, three and six degrees of freedom.

In our performance measurements we will look at the performance both with and without communication-hiding enabled. Further, we will vary our parameters:

- Degrees of freedom: one, three or six degrees
- How many subdivisions of the stiffness matrix (directly affecting the size of each subdivision of data being transferred)
- The whole matrix size: number of non-zeroes, as a result of 128^3 (~2 mill.), 256^3 (~16.8 mill.), 160^3 (~4.1 mill.) or 96^3 (~0.9 mill.) nodes and the degrees of freedom (as mentioned in the first point). We have to stay within the bounds of the system memory for the largest cases.

The latter property will affected the minimum amount of subdivisions needed, in order for two subdivisions to concurrently fit into the GPU memory available. Table 1 shows the properties of the stiffness matrices generated, and the space consumed by the non-zero elements alone.

Table 1. Stiffness matrix properties

Degrees of freedom	Number of nodes	Number of non-zeroes	Size of non-zeroes in the stiffness matrix, in GB
1	128^3	55,742,968 x 1	0.42
1	256^3	449,455,096 x 1	3.39
3	128^3	55,742,968 x 9	3.74
3	160^3	109,215,352 x 9	7.32
6	96^3	23,393,656 x 36	6.27
6	128^3	55,742,968 x 36	14.95

Since SpMV is a highly memory bound type of application we can estimate the highest performance theoretically possible to achieve on a certain piece of hardware if we know its memory bandwidth. Of course, we also have to take the algorithm’s bytes-per-flop property into consideration, for this estimation. By counting the number of double precision operations in our routine, and how many reads we have of doubles, we can find the absolute upper limit for the performance that can be achieved with this routine, for a particular hardware device. The calculation is done as follows:

$$\text{FLOP} / \text{FLIO} \times \text{Bandwidth} = \text{Performance theoretically achievable} \quad (1)$$

Here, ‘FLOP’ is number of floating point operations and ‘FLIO’ is read operations in bytes. For the AMD card we get

$$2 \text{ FLOP} / 16 \text{ byte} \times 288 \text{ GB/s} = 36 \text{ GFLOP/s} \quad (2)$$

$$18 \text{ FLOP} / 96 \text{ byte} \times 288 \text{ GB/s} = 54 \text{ GFLOP/s} \quad (3)$$

$$72 \text{ FLOP} / 336 \text{ byte} \times 288 \text{ GB/s} = 61.71 \text{ GFLOP/s} \quad (4)$$

where (2) is for the one DOF SpMV kernel, (3) is for the three DOF SpMV kernel, and (4) is for the 6 DOF SpMV kernel. These serves as the maximum limits possible to achieve, intra-GPU.

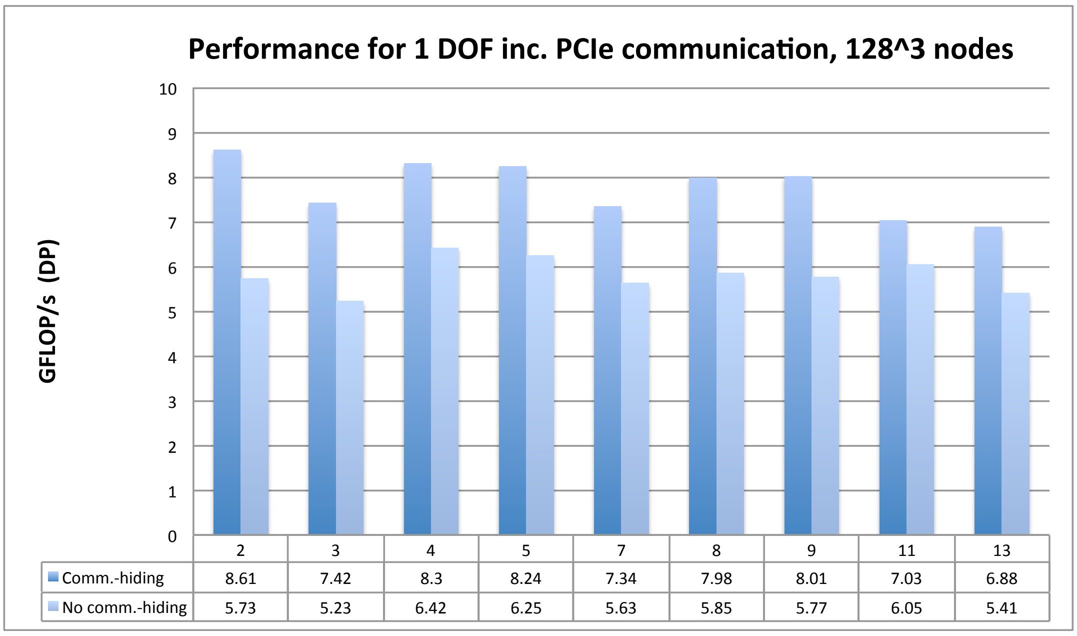
For all benchmarking the GPU in use is ‘AMD RADEON HD7970 GHz Edition’, of the ‘Tahiti’ architecture. This is connected on a PCI Express 2.0 bus. All parallel work is configured to use all physical compute units available intra-GPU, and the largest possible number of threads per compute unit (or work-group). In this case, we have 32 ‘gangs’, each of 256 ‘workers’. The processor is an Intel Nehalem i7-920. For OpenACC the PGI Accelerator C/C++ Workstation compiler release 14.3 was used, with latest AMD GPU drivers (as of April 25th 2014; Catalyst v.14.4). We used AMD CodeXL v.1.3 for GPU profiling.

Results

As a comparison we have measured the serial version, for each of the three degrees of freedom. Number of nodes is set to 128^3 for all cases, and we use ‘g++’ with the ‘–O2’ optimization flag:

- 1 DOF: 1.08 GFLOP/s
- 3 DOF: 2.67 GFLOP/s
- 6 DOF: 3.20 GFLOP/s

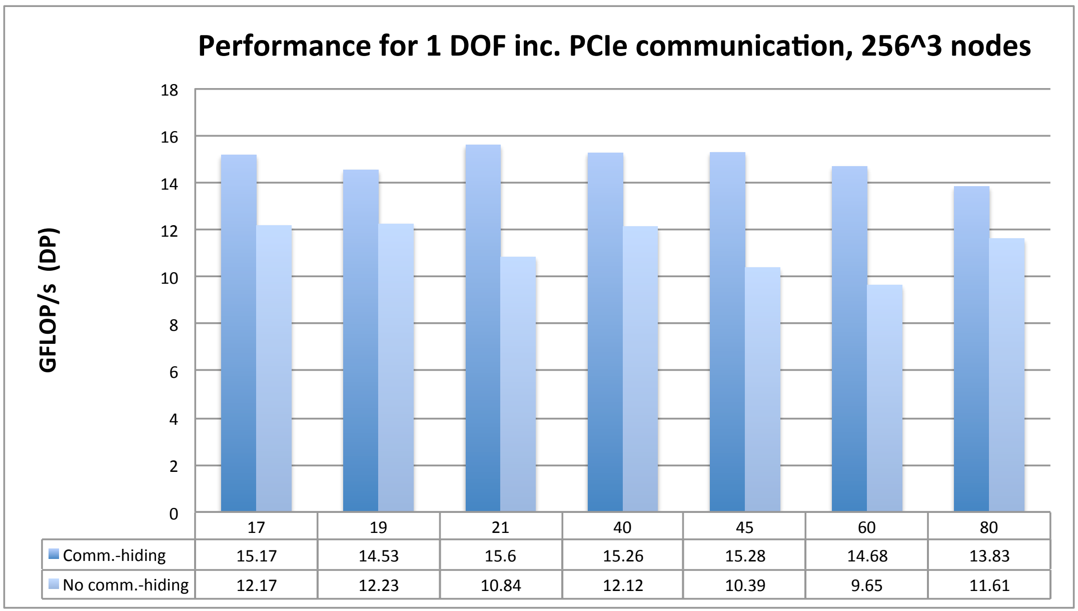
When running on the GPU, all cases inform us that we clearly benefit from the communication-hiding. *Note that the x-axis in all figures below is the amount of subdivisions applied.* In Fig. 3, the size of data is fairly small. We notice a bit varying performance. The small amount of data cannot keep the GPU busy in a sustained manner, so performance is limited.



	2	3	4	5	7	8	9	11	13
Comm.-hiding	8.61	7.42	8.3	8.24	7.34	7.98	8.01	7.03	6.88
No comm.-hiding	5.73	5.23	6.42	6.25	5.63	5.85	5.77	6.05	5.41

Figure 3. Results for 1 DOF, 128^3 nodes.

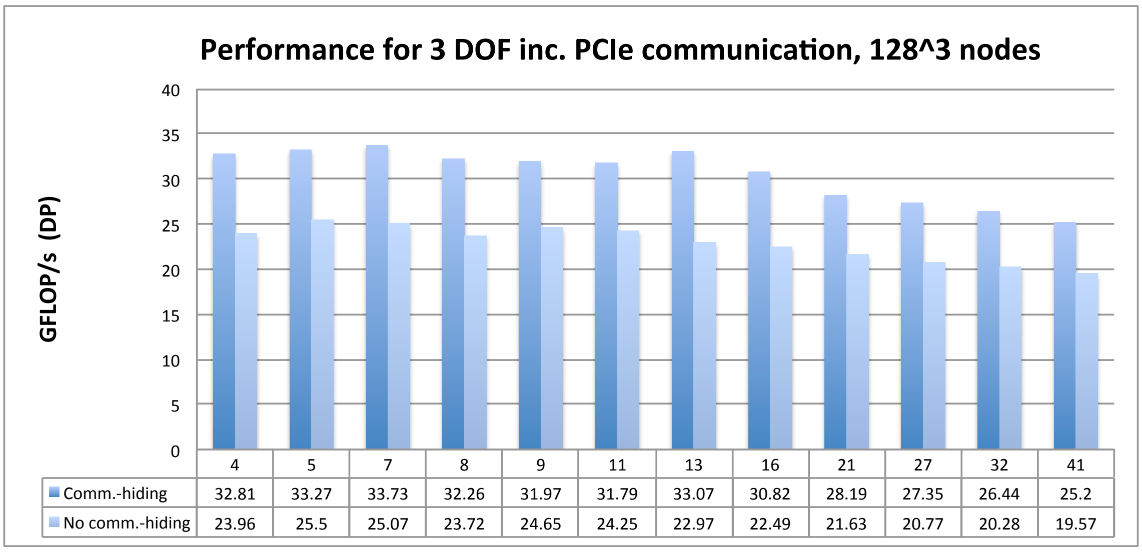
In Fig. 4 we have drastically increased the number of nodes, and as a consequence the number of non-zero elements. The GPU is kept busy to a higher degree. The sparse data does not reside blocked (limiting coalesced reads to GPU memory), and the usual high byte-to-flop ratio limits the performance. The in-GPU bandwidth cannot be exploited to a high degree.



	17	19	21	40	45	60	80
Comm.-hiding	15.17	14.53	15.6	15.26	15.28	14.68	13.83
No comm.-hiding	12.17	12.23	10.84	12.12	10.39	9.65	11.61

Figure 4. Results for 1 DOF, 256^3 nodes.

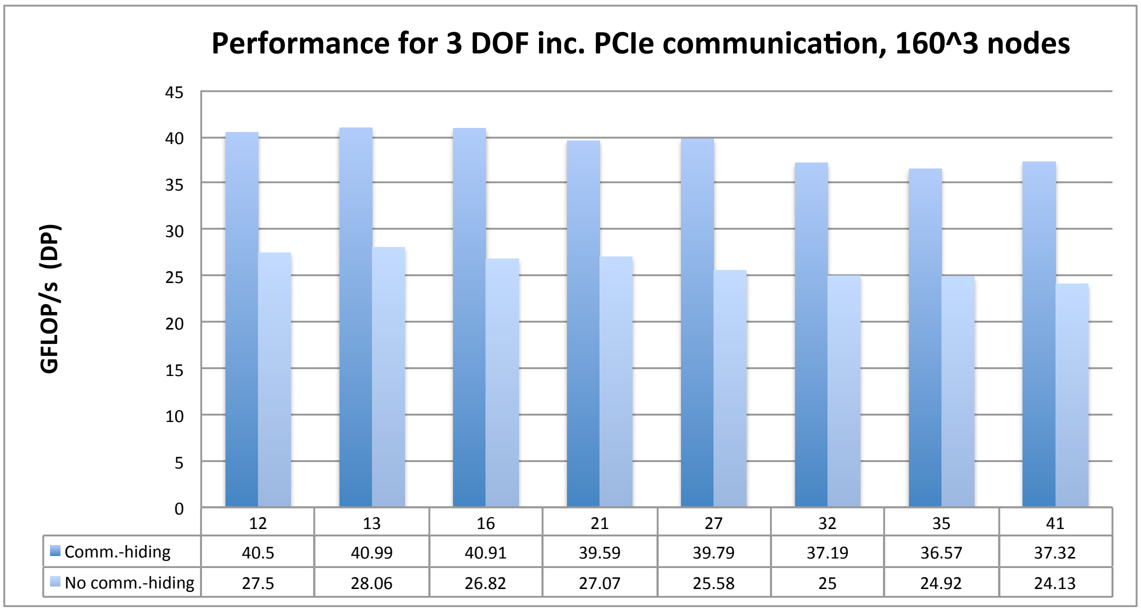
In Fig. 5 we move on to a higher degree of freedom. The data in the sparse matrix is here in 3 x 3 blocks. This gives the possibility of coalesced reads on-GPU-card, and thus a better utilization of the bandwidth the card offers. We see much improved performance. Up to about 13 subdivisions the performance is quite stable, then up to 41 subdivisions it gradually drops with several GFLOP/s.



	4	5	7	8	9	11	13	16	21	27	32	41
Comm.-hiding	32.81	33.27	33.73	32.26	31.97	31.79	33.07	30.82	28.19	27.35	26.44	25.2
No comm.-hiding	23.96	25.5	25.07	23.72	24.65	24.25	22.97	22.49	21.63	20.77	20.28	19.57

Figure 5. Results for 3 DOF, 128^3 nodes.

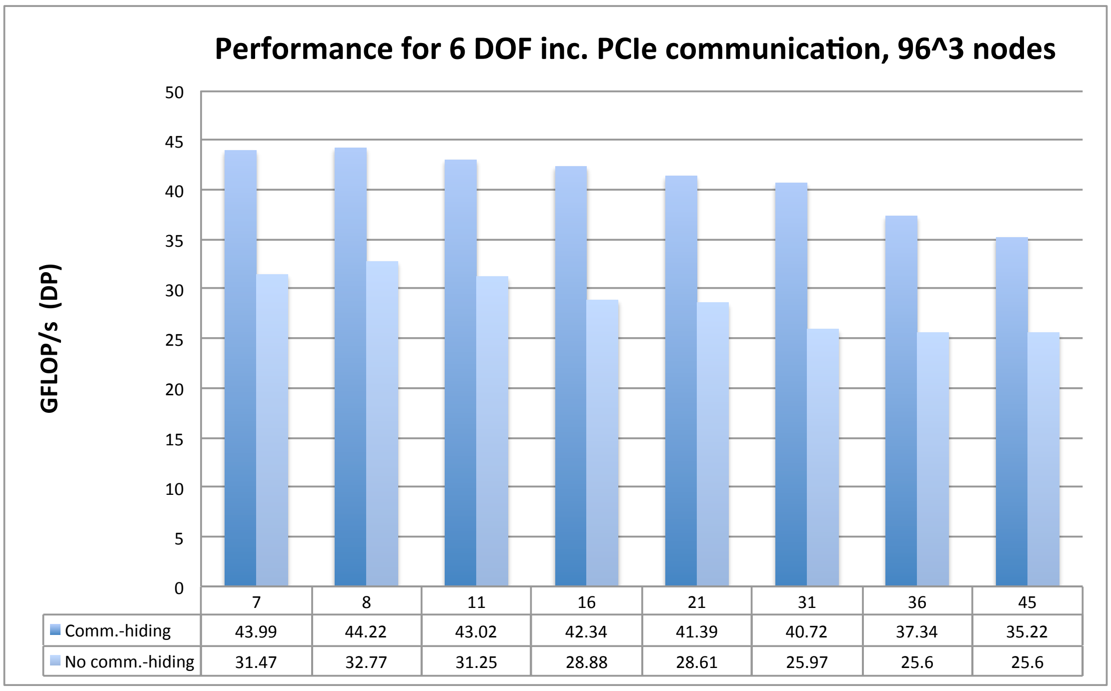
In Fig. 6., with more nodes, each subdivision number will give larger subdivisions. The performance is stable from 12 up to 41 subdivisions. Performance with communication-hiding is significantly higher than that of Fig. 5. Total data-size is about twice, and the data amount will possibly keep the GPU better utilized than in Fig. 5.



	12	13	16	21	27	32	35	41
Comm.-hiding	40.5	40.99	40.91	39.59	39.79	37.19	36.57	37.32
No comm.-hiding	27.5	28.06	26.82	27.07	25.58	25	24.92	24.13

Figure 6. Results for 3 DOF, 160^3 nodes.

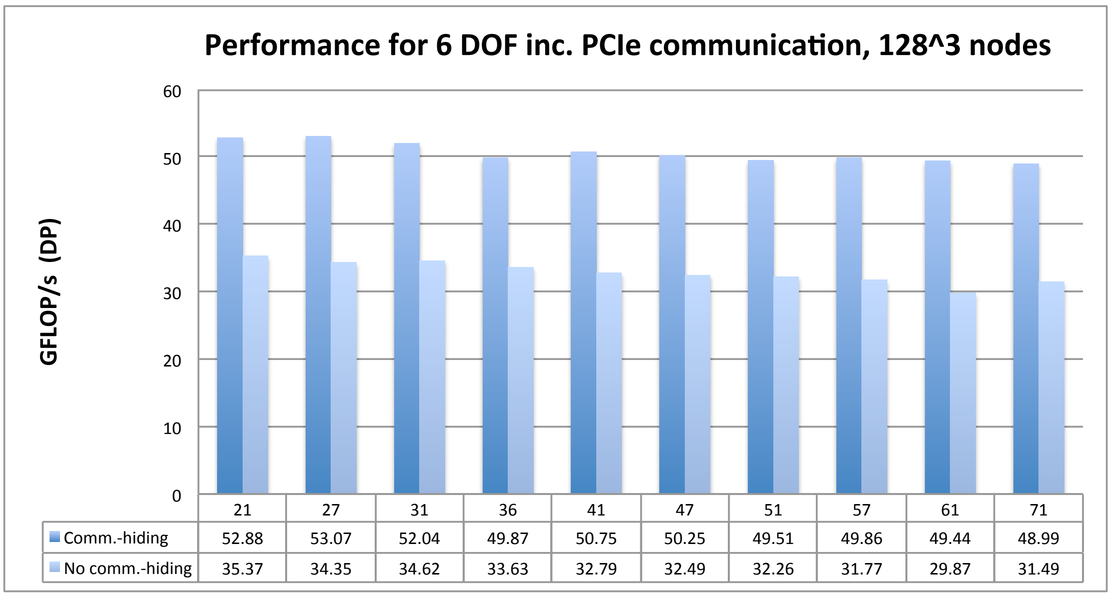
In Fig. 7 we have six degrees of freedom. That means even better conditions for coalesced reads of data on-GPU, and this is reflected in a performance jump compared to Fig. 6 and 5. We also observe a performance decrease as number of subdivisions increases.



	7	8	11	16	21	31	36	45
Comm.-hiding	43.99	44.22	43.02	42.34	41.39	40.72	37.34	35.22
No comm.-hiding	31.47	32.77	31.25	28.88	28.61	25.97	25.6	25.6

Figure 7. Results for 6 DOF, 96^3 nodes.

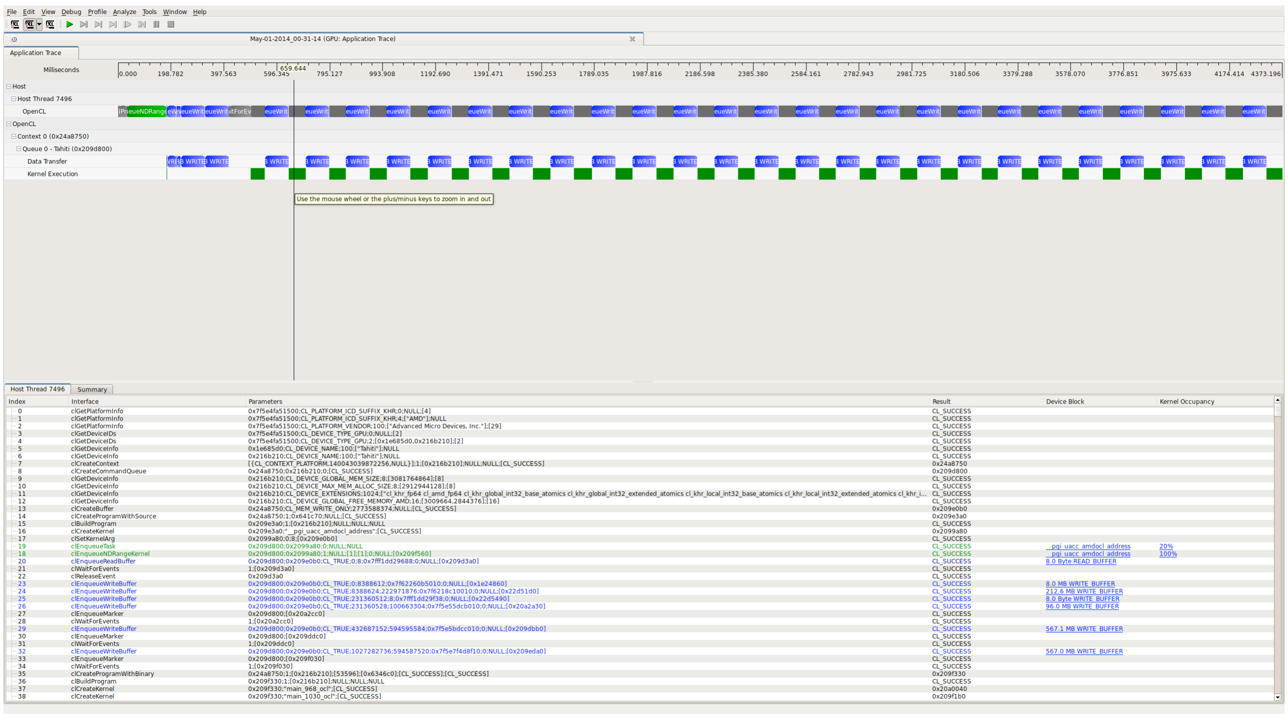
In Fig. 8 the combined non-zero size is at the largest, close to 15 GB. Combined with the large amount of coalesced reads, we here observe the highest performance. This applies to both with and without communication-hiding, when "competing" with the other alternatives of parameters and their associated performance. Here, we also observe the most significant gain of performance, when enabeling communication-hiding; up to ~65.7% increased performance for 61 subdivisions.



	21	27	31	36	41	47	51	57	61	71
Comm.-hiding	52.88	53.07	52.04	49.87	50.75	50.25	49.51	49.86	49.44	48.99
No comm.-hiding	35.37	34.35	34.62	33.63	32.79	32.49	32.26	31.77	29.87	31.49

Figure 8. Results for 6 DOF, 128^3 nodes.

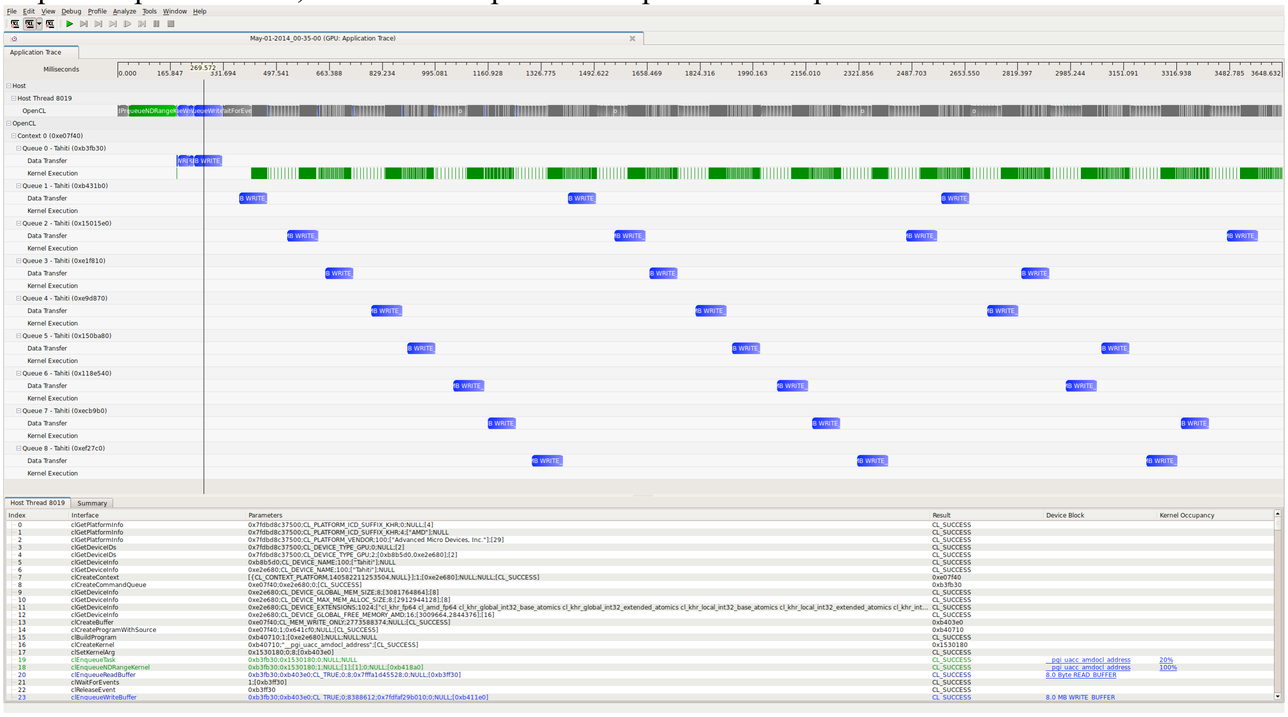
When using the profiler, we can get better insight into what is actually happening on the GPU. Particularly, how the asynchronous and synchronous data transfers behave, and differ. It gives us a "window" to see how the communicatin-hiding actually works. In the profiler, the timeline stretches horizontally. In both Fig. 9 and 10 the blue blocks illustrates the transfers of the subdivisions, while green blocks illustrates GPU computation, i.e. the SpMV being executed, in our case.


File Edit View Debug Profile Analyze Tools Window Help
May-01-2014_00-31-14 (GPU: Application Trace)
Application Trace
Milliseconds
Host
Host Thread 7496
OpenCL
Context 0 (0x24a8750)
Queue 0 - Tahiti (0x209d800)
Data Transfer
Kernel Execution
Use the mouse wheel or the plus/minus keys to zoom in and out
Host Thread 7496
Summary

Index	Interface	Parameters	Result	Device Block	Kernel Occupancy
0	clGetPlatformInfo	0x7f5e4fa51500;CL_PLATFORM_ICD_SUFFIX_KHR;0;NULL;[4]	CL_SUCCESS		
1	clGetPlatformInfo	0x7f5e4fa51500;CL_PLATFORM_ICD_SUFFIX_KHR;4;["AMD"];NULL	CL_SUCCESS		
2	clGetPlatformInfo	0x7f5e4fa51500;CL_PLATFORM_VENDOR;100;["Advanced Micro Devices, Inc."];[29]	CL_SUCCESS		
3	clGetDeviceIDs	0x7f5e4fa51500;CL_DEVICE_TYPE_GPU;0;NULL;[2]	CL_SUCCESS		
4	clGetDeviceIDs	0x7f5e4fa51500;CL_DEVICE_TYPE_GPU;2;[0x1e685d0,0x216b210];[2]	CL_SUCCESS		
5	clGetDeviceInfo	0x1e685d0;CL_DEVICE_NAME;100;["Tahiti"];NULL	CL_SUCCESS		
6	clGetDeviceInfo	0x216b210;CL_DEVICE_NAME;100;["Tahiti"];NULL	CL_SUCCESS		
7	clCreateContext	[{CL_CONTEXT_PLATFORM,140043039872256,NULL}];1;[0x216b210];NULL;NULL;[CL_SUCCESS]	0x24a8750		
8	clCreateCommandQueue	0x24a8750;0x216b210;0;[CL_SUCCESS]	0x209d800		
9	clGetDeviceInfo	0x216b210;CL_DEVICE_GLOBAL_MEM_SIZE;8;[3081764864];[8]	CL_SUCCESS		
10	clGetDeviceInfo	0x216b210;CL_DEVICE_MAX_MEM_ALLOC_SIZE;8;[2912944128];[8]	CL_SUCCESS		
11	clGetDeviceInfo	0x216b210;CL_DEVICE_EXTENSIONS;1024;["cl_khr_fp64 cl_amd_fp64 cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics cl_khr_i...	CL_SUCCESS		
12	clGetDeviceInfo	0x216b210;CL_DEVICE_GLOBAL_FREE_MEMORY_AMD;16;[3009664,2844376];[16]	CL_SUCCESS		
13	clCreateBuffer	0x24a8750;CL_MEM_WRITE_ONLY;2773588374;NULL;[CL_SUCCESS]	0x209e0b0		
14	clCreateProgramWithSource	0x24a8750;1;0x641c70;NULL;[CL_SUCCESS]	0x209e3a0		
15	clBuildProgram	0x209e3a0;1;[0x216b210];NULL;NULL;NULL	CL_SUCCESS		
16	clCreateKernel	0x209e3a0;"__pgi_uacc_amdocl_address";[CL_SUCCESS]	0x2099a80		
17	clSetKernelArg	0x2099a80;0;8;[0x209e0b0]	CL_SUCCESS		
19	clEnqueueTask	0x209d800;0x2099a80;0;NULL;NULL	CL_SUCCESS	__pgi_uacc_amdocl_address	20%
18	clEnqueueNDRangeKernel	0x209d800;0x2099a80;1;NULL;[1];[1];0;NULL;[0x209f560]	CL_SUCCESS	__pgi_uacc_amdocl_address	100%
20	clEnqueueReadBuffer	0x209d800;0x209e0b0;CL_TRUE;0;8;0x7fff1dd29688;0;NULL;[0x209d3a0]	CL_SUCCESS	8.0 Byte READ_BUFFER	
21	clWaitForEvents	1;[0x209d3a0]	CL_SUCCESS		
22	clReleaseEvent	0x209d3a0	CL_SUCCESS		
23	clEnqueueWriteBuffer	0x209d800;0x209e0b0;CL_TRUE;0;8388612;0x7f62260b5010;0;NULL;[0x1e24860]	CL_SUCCESS	8.0 MB WRITE_BUFFER	
24	clEnqueueWriteBuffer	0x209d800;0x209e0b0;CL_TRUE;8388624;222971876;0x7f6218c10010;0;NULL;[0x22d51d0]	CL_SUCCESS	212.6 MB WRITE_BUFFER	
25	clEnqueueWriteBuffer	0x209d800;0x209e0b0;CL_TRUE;231360512;8;0x7fff1dd29f38;0;NULL;[0x22d5490]	CL_SUCCESS	8.0 Byte WRITE_BUFFER	
26	clEnqueueWriteBuffer	0x209d800;0x209e0b0;CL_TRUE;231360528;100663304;0x7f5e55dcb010;0;NULL;[0x20a2a30]	CL_SUCCESS	96.0 MB WRITE_BUFFER	
27	clEnqueueMarker	0x209d800;[0x20a2cc0]	CL_SUCCESS		
28	clWaitForEvents	1;[0x20a2cc0]	CL_SUCCESS		
29	clEnqueueWriteBuffer	0x209d800;0x209e0b0;CL_TRUE;432687152;594595584;0x7f5e5bdcc010;0;NULL;[0x209dbb0]	CL_SUCCESS	567.1 MB WRITE_BUFFER	
30	clEnqueueMarker	0x209d800;[0x209ddc0]	CL_SUCCESS		
31	clWaitForEvents	1;[0x209ddc0]	CL_SUCCESS		
32	clEnqueueWriteBuffer	0x209d800;0x209e0b0;CL_TRUE;1027282736;594587520;0x7f5e7f4d8f10;0;NULL;[0x209eda0]	CL_SUCCESS	567.0 MB WRITE_BUFFER	
33	clEnqueueMarker	0x209d800;[0x209f030]	CL_SUCCESS		
34	clWaitForEvents	1;[0x209f030]	CL_SUCCESS		
35	clCreateProgramWithBinary	0x24a8750;1;[0x216b210];[53596];[0x6346c0];[CL_SUCCESS];[CL_SUCCESS]	0x209f330		
36	clBuildProgram	0x209f330;1;[0x216b210];NULL;NULL;NULL	CL_SUCCESS		
37	clCreateKernel	0x209f330;"main_968_ocl";[CL_SUCCESS]	0x20a0040		
38	clCreateKernel	0x209f330;"main_1030_ocl";[CL_SUCCESS]	0x209f1b0		

Figure 9. Profiling when communication-hiding is disabled, all memory transfers are synchronous.

One can easily see how the data-transfer command is blocking, or synchronous, in Fig. 9. No other GPU related work can happen while the synchronous call is being executed. In Fig. 10 we clearly see the effect of using asynchronous, non-blocking, calls. While data is in-flight to the GPU memory, the SpMV routine is executed on a set of data already in-memory. This contributes to the improved performance, for the whole parameter-space we set up.


File Edit View Debug Profile Analyze Tools Window Help
May-01-2014_00-35-00 (GPU: Application Trace)
Application Trace
Milliseconds
Host
Host Thread 8019
OpenCL
Context 0 (0xe07f40)
Queue 0 - Tahiti (0xb3fb30)
Queue 1 - Tahiti (0xb431b0)
Queue 2 - Tahiti (0x15015e0)
Queue 3 - Tahiti (0xe1f810)
Queue 4 - Tahiti (0xe9d870)
Queue 5 - Tahiti (0x150ba80)
Queue 6 - Tahiti (0x118e540)
Queue 7 - Tahiti (0xecb9b0)
Queue 8 - Tahiti (0xef27c0)
Data Transfer
Kernel Execution
Host Thread 8019
Summary

Index	Interface	Parameters	Result	Device Block	Kernel Occupancy
0	clGetPlatformInfo	0x7fdbd8c37500;CL_PLATFORM_ICD_SUFFIX_KHR;0;NULL;[4]	CL_SUCCESS		
1	clGetPlatformInfo	0x7fdbd8c37500;CL_PLATFORM_ICD_SUFFIX_KHR;4;["AMD"];NULL	CL_SUCCESS		
2	clGetPlatformInfo	0x7fdbd8c37500;CL_PLATFORM_VENDOR;100;["Advanced Micro Devices, Inc."];[29]	CL_SUCCESS		
3	clGetDeviceIDs	0x7fdbd8c37500;CL_DEVICE_TYPE_GPU;0;NULL;[2]	CL_SUCCESS		
4	clGetDeviceIDs	0x7fdbd8c37500;CL_DEVICE_TYPE_GPU;2;[0xb8b5d0,0xe2e680];[2]	CL_SUCCESS		
5	clGetDeviceInfo	0xb8b5d0;CL_DEVICE_NAME;100;["Tahiti"];NULL	CL_SUCCESS		
6	clGetDeviceInfo	0xe2e680;CL_DEVICE_NAME;100;["Tahiti"];NULL	CL_SUCCESS		
7	clCreateContext	[{CL_CONTEXT_PLATFORM,140582211253504,NULL}];1;[0xe2e680];NULL;NULL;[CL_SUCCESS]	0xe07f40		
8	clCreateCommandQueue	0xe07f40;0xe2e680;0;[CL_SUCCESS]	0xb3fb30		
9	clGetDeviceInfo	0xe2e680;CL_DEVICE_GLOBAL_MEM_SIZE;8;[3081764864];[8]	CL_SUCCESS		
10	clGetDeviceInfo	0xe2e680;CL_DEVICE_MAX_MEM_ALLOC_SIZE;8;[2912944128];[8]	CL_SUCCESS		
11	clGetDeviceInfo	0xe2e680;CL_DEVICE_EXTENSIONS;1024;["cl_khr_fp64 cl_amd_fp64 cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics cl_khr_int...	CL_SUCCESS		
12	clGetDeviceInfo	0xe2e680;CL_DEVICE_GLOBAL_FREE_MEMORY_AMD;16;[3009664,2844376];[16]	CL_SUCCESS		
13	clCreateBuffer	0xe07f40;CL_MEM_WRITE_ONLY;2773588374;NULL;[CL_SUCCESS]	0xb403e0		
14	clCreateProgramWithSource	0xe07f40;1;0x641cf0;NULL;[CL_SUCCESS]	0xb40710		
15	clBuildProgram	0xb40710;1;[0xe2e680];NULL;NULL;NULL	CL_SUCCESS		
16	clCreateKernel	0xb40710;"__pgi_uacc_amdocl_address";[CL_SUCCESS]	0x1530180		
17	clSetKernelArg	0x1530180;0;8;[0xb403e0]	CL_SUCCESS		
19	clEnqueueTask	0xb3fb30;0x1530180;0;NULL;NULL	CL_SUCCESS	__pgi_uacc_amdocl_address	20%
18	clEnqueueNDRangeKernel	0xb3fb30;0x1530180;1;NULL;[1];[1];0;NULL;[0xb418a0]	CL_SUCCESS	__pgi_uacc_amdocl_address	100%
20	clEnqueueReadBuffer	0xb3fb30;0xb403e0;CL_TRUE;0;8;0x7fffa1d45528;0;NULL;[0xb3ff30]	CL_SUCCESS	8.0 Byte READ_BUFFER	
21	clWaitForEvents	1;[0xb3ff30]	CL_SUCCESS		
22	clReleaseEvent	0xb3ff30	CL_SUCCESS		
23	clEnqueueWriteBuffer	0xb3fb30;0xb403e0;CL_TRUE;0;8388612;0x7fdfaf29b010;0;NULL;[0xb411e0]	CL_SUCCESS	8.0 MB WRITE_BUFFER	

Figure 10. Profiling when communication-hiding is enabled. All memory transfers except the initial one is asynchronous.

Conclusions

When we consider the peak performance possible intra-GPU for these data-intensive algorithms and with the hardware used, we come impressively close when applying our communication-hiding. The closest is achieved with 6 degrees of freedom and 128^3 number of nodes; 53.07 GFLOP/s is achieved - whereas the peak possible intra-GPU is 61.71 GFLOP/s. On the opposite side, we are the farthest away from the peak possible when having one degree of freedom and 128^3 nodes.
We have seen how SpMV can achieve improved performance by communication-hiding – or, overlapping of computation and data-transfers. This is of direct relevance for, for instance, block Krylov solvers, and makes streaming over the PCI express bus more efficient. This is useful especially when the amount of non-zeroes is so large that all data cannot fit the GPU memory.
The communication-hiding scheme becomes more efficient when the amount of subdivisions is high enough so that the first synchronous data-transfer performed in order to buffer does not dominate the total time used for computation and communication. Also, there is a "sweet-spot" where the computation done over each subdivision takes as similar amount of time as possible as the current subdivision being transferred takes to transfer. This controls the rate of efficiency.
It was observed a performance improvement over the whole parameter-space tested. The performance improvement increased as the total amount of non-zero data increased and the possibility of intra-GPU coalesced reads increased. This means that the more efficiently the GPU memory sub-system is utilized, the better effect is observed from the communication-hiding. At the very best, we found an increase in performance of ~65.7%. This underlines the importance of applying such techniques in simulations, when it is suitable with the algorithmic structure of the problem in relation to the underlying computer architecture – as examples here in relation to our SpMV; block Krylov and attached discrete GPUs with a separate memory hierarchy.

Future work

A thorough study of the performance portability is of importance. From the start, the tools selected and the use of them were done so to maintain a high possibility of performance portability. One could, of course, want to run the routines on a future GPU architecture, from the same or a different vendor. Doing such should have as little burden on the developer as possible.
Automatically finding the optimal parameters, for a given problem, is of interest. That is to say, the number of subdivisions that gives the best communication-hiding (matrix size and degrees of freedom already determined by the problem).
As part of a MPI multi-node solver, communication-hiding can be applied at several levels, or layers. Multi-layer communication-hiding is of interest for our research.

References

Sato, Y. and Okuda, H. (2008) Grid-Aided Stochastic Finite Element Using Ninf-G, *8th World Congress on Computational Mechanics (WCCM8)*.

Bell, N. and Garland, M. (2008) Efficient sparse matrix-vector multiplication in CUDA. Vol. 2. No. 5. *NVIDIA Technical Report NVR-2008-004*, NVIDIA Corporation.

Saad, Y. (2003) Iterative methods for sparse linear systems. Siam.

Vuduc, R. W. (2003) Automatic performance tuning of sparse matrix kernels, *Dissertation*, University of California.