Technical Report, CALDGEMM and HPL

David Rohr, Matthias Kretz, Matthias Bach
Frankfurt Institute for Advanced Studies,
University of Frankfurt,
Germany

December 10, 2010

Abstract

The LOEWE-CSC cluster at the Goethe University of Frankfurt is a heterogeneous compute-cluster employing AMD Magny-Cours CPUs and Cypress GPUs. A fast DGEMM algorithm was developed which is able to use both the GPU and all CPU cores to the full extent. It is shown that the transfer time to and from the GPU can be almost completely hidden by the calculation time. Additionally, a dynamic scheduler ensures full utilization of the processors. The HPL [Hpl] benchmark was parallelized, vectorized, and adapted to utilize the fast DGEMM. Additionally, alignment optimizations were applied and the lookahead algorithm, which hides the communication time, was completely rewritten. The GPU kernel itself, without the transfer to and from the GPU, is able to achieve 497 GFlop/s which corresponds to to 90.93% of the GPU peak performance. To our knowledge there exists no faster kernel implementation. The combined CPU/GPU DGEMM achieves 623 GFlop/s which is 83.63% of the accumulated peak performance. The multi-node Linpack implementation is able to achieve 70% of the theoretical performance. It is demonstrated that this performances scales linearly to hundreds of nodes.

Contents

1 Introduction to Linpack & DGEMM

1.1 Linpack

The Linpack benchmark is one of the widest used benchmark to measure the performance of supercomputers and builds the basis for the Top500 Supercomputer List [Top1]. The benchmark solves a dense system of linear equations. The rules include that the algorithm used to solve the system must be of complexity $O(N^3)$ where N is the matrix size. This ensures that the implementation of the algorithm can be adapted to perform well on various supercomputers. By fixing the algorithm itself a fair comparison is allowed for.

In general, it can be shown that solving a system of linear equations can be reduced to matrix multiplication. The Strassen-Algorithm [Str] can perform matrix multiplication in $O(N^{\log_2 7})$. This can be even improved to $O(N^{2.375477})$ as shown in [Cop1+]. However, the latter one has no application on HPC[1], as the constants appearing are too big. In the following only the naive $O(N^3)$ matrix multiplication algorithm will be used and only its implementation improved.

1.2 High Performance Linpack (HPL)

The High Performance Linpack [Hpl] is an implementation of the Linpack benchmark provided by the University of Tennessee and the University of Colorado. It implements the benchmark by performing an LU factorization with row partial pivoting and solving the triangular system afterwards. This is done using a nested recursion. Let $A \in M_{n,n+1}$ be a $n \times (n+1)$ matrix. Each iteration in the outermost loop factors a panel of nb columns. This is done by LU-factorizing the top-left $nb \times nb$ submatrix using an arbitrary solver. Solving this submatrix will be referred to as "Factorization" hereafter. After the panel has been factorized, it must be broadcast to all nodes that take part in the computation. Then the row-swaps of the pivoting process must be repeated on the trailing submatrix. The U-matrix must then be solved against the triangular system and also be broadcast. This is implemented by first creating and broadcasting[2] the U-matrix and then replicating the solve process on each node. In the last step, the C-matrix must be updated, by adding the columns of U in the way, that zeroed out all rows in the former L-matrix. This is done by subtracting the matrix-product $L \cdot U$. Then the next iteration starts on the matrix C. Fig. 1 shows all relevant submatrices.

All steps except for the final matrix multiplication can be performed in $O(N^2)$.[3] The $O(N^3)$ matrix multiplication therefore is the time critical part of the HPL and will thus be analyzed in more detail.

1.3 DGEMM

DGEMM is the abbreviation for "Double-Precision Generalized Matrix Multiplication". It computes the generalized matrix-product $C' = \alpha \cdot \mathrm{op}(A)\mathrm{op}(B) + \beta \cdot C$ where C' is overwritten onto C, α and β are scalars, A is an $m \times k$-matrix, and B is an $k \times n$ matrix. $\mathrm{op}(A)$ can be A or A^t. For the time being this parameter shall be ignored. A and B will be assumed transposed

[1] High Performance Computing.

[2] The broadcast is performed using a spread and roll algorithm as described in [Hpl].

[3] The matrix multiplication is at least $O(N^2)$ as it must calculate at least N^2 distinct values. This already shows how LU-factorization can be reduced to matrix multiplication.

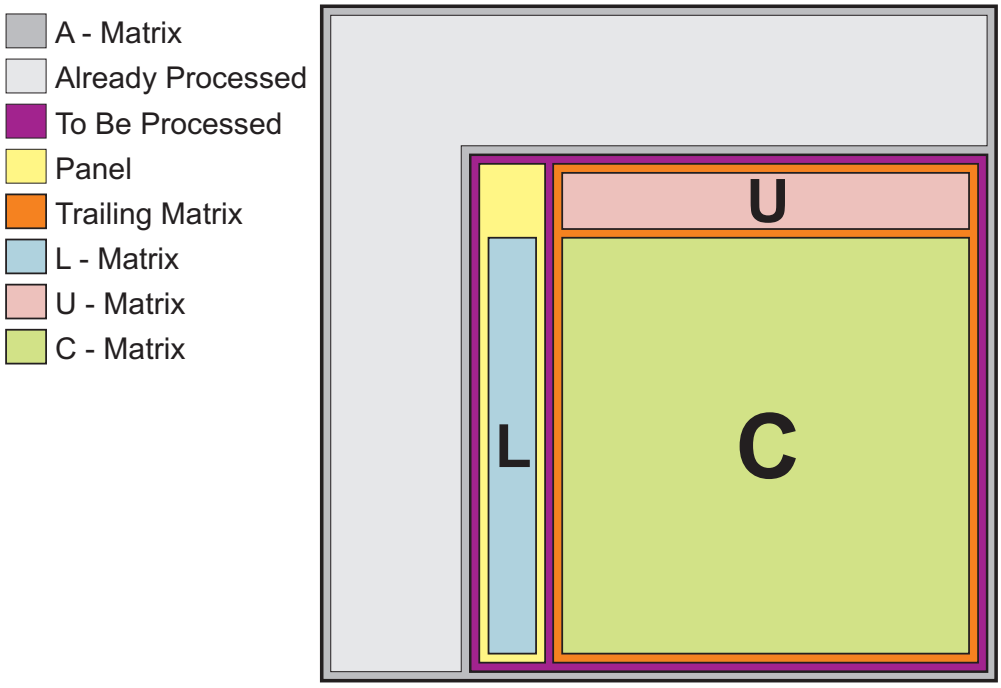

Figure 1: Submatrices in HPL

or not transposed, whichever is suited better. Section 2.5.1 will show how transposed matrices are treated in CALDGEMM (See Section 2).

DGEMM is part of the "Basic Linear Algebra System" (BLAS), which is a common interface for matrix / vector operations. HPL can use different BLAS libraries. For the Linpack at the LOEWE-CSC [Loe] the GotoBLAS2 library [Tac] was used. It uses multiple blocking levels[4] and was specifically optimized to respect page boundaries and minimize TLB[5] faults.

For AMD GPUs the ACML-GPU library is available.[6] However, it was decided to come up with an approach written from scratch because the ACML-GPU was unable to utilize the GPU to the extent that was desired.

2 CALDGEMM

CALDGEMM is a library for GPU based DGEMM that was developed at the University of Frankfurt. Currently it can run on AMD Cypress hardware. However, it should be easily adaptable to other GPUs like the NVIDIA Fermi or the upcoming Intel Knights Corner [Int4] accelerators. It consists of two parts: The kernel that executes the main matrix multiplication on the GPU and the framework that handles the DMA transfer as well as data pre- and postprocessing. In the first approach the kernel from the double_matmult example of the AMD Stream SDK [Amd1] was adapted. The framework itself was written from scratch.

[4] See Section 2.4.1.
[5] Translation Lookaside Buffer.
[6] AMD Core Math Library [Amd4].

2.1 GPU based DGEMM

In a first step the scalars and the addition in the DGEMM shall be ignored, but only the matrix-product $C = AB$ be calculated. When not stated otherwise explicitly, $k = 1024$ is assumed. The justification for this will come up throughout this chapter, when at some points different k values are analyzed. The BLAS interface supports both, col-major and row-major matrices.[7] Switching from col- to row-major and vice versa corresponds to the transversion $A \cdot B \Rightarrow (A^t \cdot B^t)^t = B \cdot A$ so only the order of the operands needs to be changed. Therefore, in the following all matrices can be assumed row-major.

The AMD GPUs employed have memory sizes between 1 and 4 GBs. The nodes in the LOEWE-CSC cluster offer 64 gigabytes of main memory. The size of the processed matrices has the same order of magnitude. Therefore, it is not possible (or even desirable) to process the whole DGEMM operation at once on the GPU, but a pipelined streaming approach was implemented. The LOEWE-CSC is equipped with two 12-core Magny-Cours processors per node. These 24 cores provide a non negligible amount of compute power, which should be used for the DGEMM as well. For the streaming process a tiling is applied. The matrices A, B, and C are divided into sub matrices A_i, B_j, and $C_{i,j}$ whose dimensions are powers of two or multiples of higher powers of two. The $C_{i,j}$ are called tiles. The matrix-product $C = AB$ can thus be calculated as $C_{i,j} = A_i \cdot B_j$ as seen in Fig. 2.

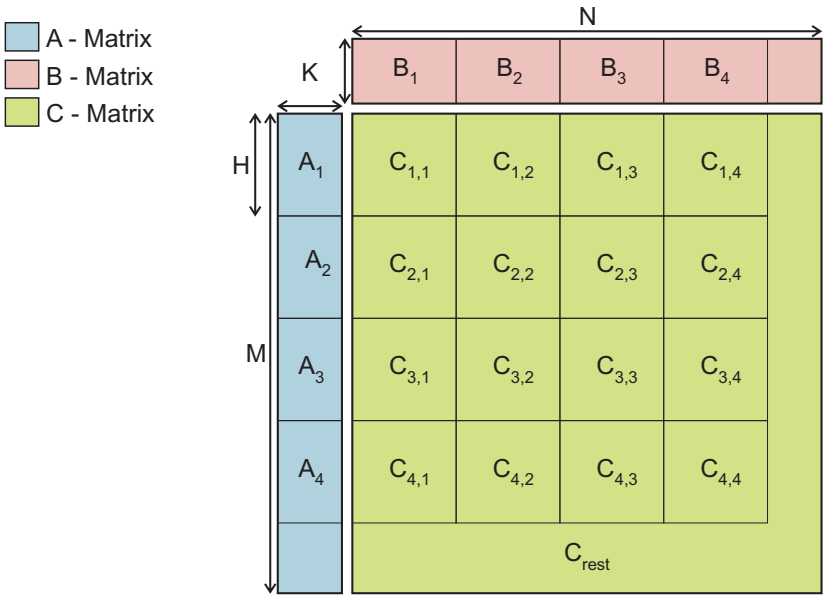

Figure 2: Splitting Matrices for Streaming DGEMM

The original matrix dimensions are not necessarily powers of two, or multiples of at least say 64. However, this does not pose a problem, as the remaining part C_{Rest} can easily be calculated by the CPU. Thus, as a further simplification $C_{i,j}$ is assumed as a square matrix of dimension h.

The best data arrangement for the GPU to read and write the data is not necessarily the arrangement used for the plain input and output matrices A_i, B_j, and $C_{i,j}$. E.g. it might be good to store multiple rows in an interleaved format. Thus pre- and postprocessing of the

[7] Col-major matrices are primarily used by Fortran programs, while C code usually implements the row-major version.

matrices is performed. These steps are called "DivideBuffer" and "MergeBuffer" respectively.[8] The DivideBuffer function can also be used to transpose the input matrices, as required by the BLAS specifications.

The first naive CALDGEMM implementation works the following way. The output matrix C is divided into two parts: one processed by the CPU and the other by the GPU. This is done in a way, such that the GPU part fulfills the above size restrictions. The CPU part will be processed by GotoBLAS. The GPU part will be streamed through the GPU. First the input matrices A_1 and B_1 are preprocessed, then transferred to the GPU. Then the GPU kernel is executed to calculate the matrix-product, which is transferred back to the host. Finally the postprocessing is performed. Then the iteration starts again with the next tile. Fig. 3 visualizes the process just described.

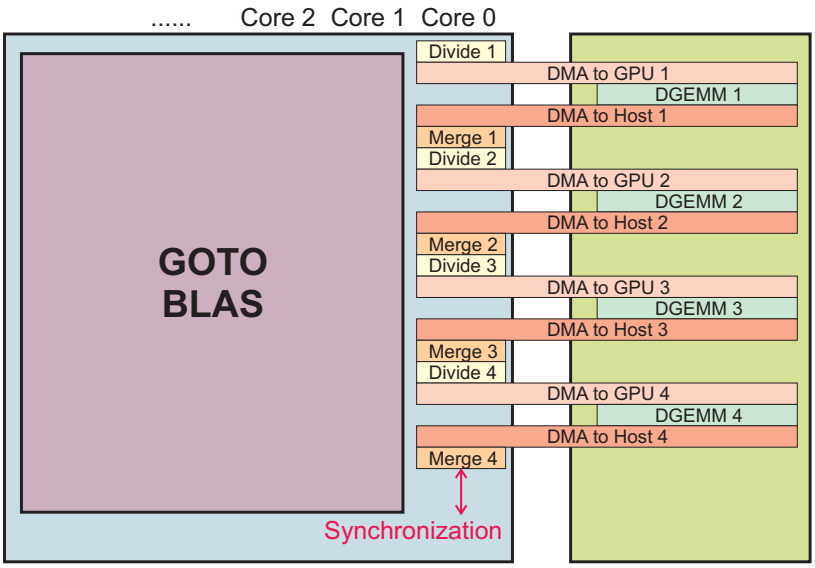

Figure 3: Process-Flow of first CALDGEMM Implementation

Fig. 3 directly reveals two issues. GPU and CPU must be synchronized well so that none of them is idling towards the end of the iteration. Thus either the required computation time must be known in advance or they must be scheduled dynamically. In addition the pre- and post-processing as well as the transfer to and from the GPU should overlap with GPU calculation.

2.2 Implementation Details

Both input and output matrices can be spread over multiple buffers. This needs to be done for certain reasons.[9] Input buffers as well as output buffers can reside in both GPU and host memory. The GPU kernel can directly read from and write data to the host memory using DMA. CALDGEMM always locates the input memory on the GPU, as the input throughput of the kernel dramatically exceeds the PCIe bandwidth. For the output memory both variants are implemented. Input data is preprocessed by DivideBuffer into page locked buffers on the

[8]The names were chosen as they are in the AMD Stream SDK sample double_matmult, which CALDGEMM was originally based on.

[9]Using multiple input buffers can reduce the register requirements (See Section 2.4.1) and alter the cache access pattern. Multiple output buffers are required for the color buffer output (See Section 2.4.1)

host of exactly the same size as the input buffers on the GPU. Those are then transferred via one large DMA transfer. The output is either directly written to the host or it is written to GPU memory and afterwards transferred to the host via one large DMA transfer. The original C matrix is never transferred to the GPU, as it is read only once. The GPU only calculates $X = \alpha \cdot AB$. The MergeBuffer routine then calculates $C' = X + \beta \cdot C$. Section 2.4.1 will explain how transposed matrices are processed. This way a full DGEMM is implemented. The parameters m, n, k, and *alpha* are stored in constant buffers on the GPU.

2.3 Combined GPU/CPU DGEMM

This section shall describe in more detail how the combined GPU/CPU DGEMM is implemented. The matrix is split in y-direction, where the upper part is processed by the GPU.[10] The part processed by the GPU is defined as $u \cdot h$ with u minimal such that $u \cdot h \geq m \cdot r$ where r is the GPU-ratio. The first naive implementation calculates the ratio $\frac{\text{MaxFlops}_{\text{GPU}}}{\text{MaxFlops}_{\text{GPU}}+\text{MaxFlops}_{\text{CPU}}}$. It is desired to rather overestimate the GPU performance. Therefore $u \cdot h \geq m \cdot r$ is required instead of taking the closest multiple of h. As Fig. 3 shows, one processing component is idling towards the end of the DGEMM. It is desired that rather the CPU idles instead of the GPU, as the GPU is the faster one.

Within each row the CPU calculates the rightmost n mod h columns. This ensures the correct size of the submatrix processed by the GPU. The GPU tiles are calculated in a loop over the vertical direction and a inner loop over the horizontal direction as can be seen in Fig. 4

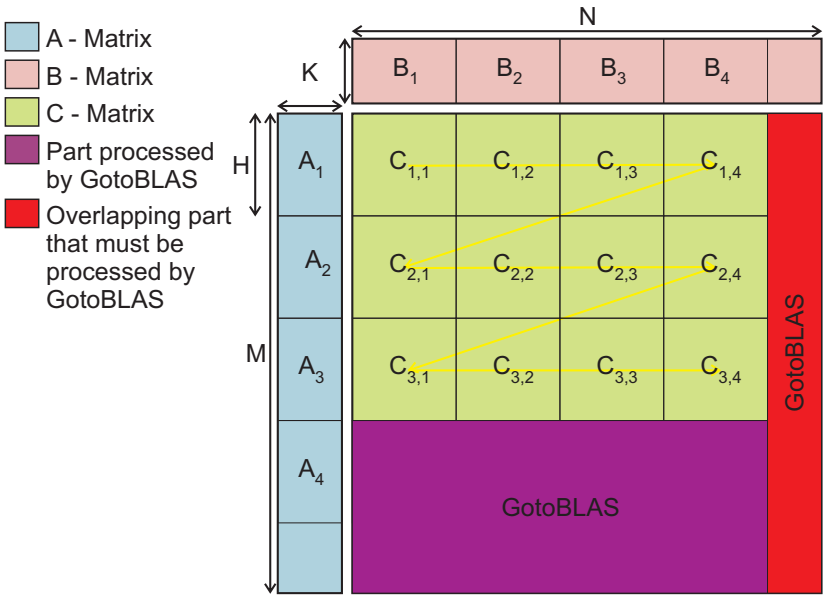

Figure 4: Running GPU/CPU DGEMM

In cases where C is not a square matrix additional logic is implemented. Splitting the matrix in an upper and a lower part does not work well with flat matrices, especially when $m \approx h$. Therefore, flat matrices are split in a left and a right part instead of upper and lower. To keep the notation simple the matrix is expected to be a square matrix for the rest of this chapter. Flat matrices are handled the appropriate way with left / right instead of up / down splitting where applicable.

[10] The reason to split the matrix in m direction is to have GPU and CPU both process consecutive memory segments.

2.3.1 Thread Pinning

The CALDGEMM implementation also requires CPU calculation, even for the GPU part of the matrix, namely the DivideBuffer and MergeBuffer functions. For all the multithreading in CALDGEMM a thread-server is implemented using pthreads synchronized by pthread-mutexes. The LOEWE-CSC employs two AMD Magny-Cours CPUs per node resulting in four independent memory controllers per node: one per CPU die.[11] It turned out that the PCIe bandwidth available depends on the memory controller responsible for the page locked memory location that is used as source and destination on the host side (See Fig. 5). One die is connected via HyperTransport to the chipset which connects the GPU via PCIe. This die's memory controller is the one to choose. For the LOEWE-CSC this is the first die (die 0).

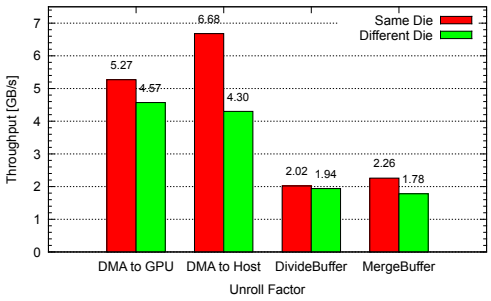

Figure 5: Dependency of PCIe bandwidth on CPU die

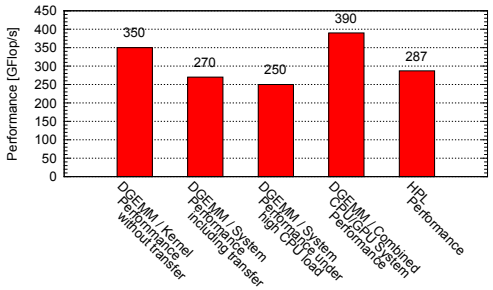

Figure 6: Performance of first CALDGEMM implementation

All GPU related memory is thus allocated on DRAM connected to die 0. To allow fast access for the DivideBuffer and MergeBuffer functions, the thread executing those, is pinned to core 0 on die 0 as well.

As 24 cores are available using $t+1$ threads for the GPU part leaves $23-t$ threads available for GotoBLAS. [12] (When multithreading CALDGEMM later all $t+1$ threads used for the GPU will be pinned to cores 0 to t.) To avoid congestion on the CPU cores the GotoBLAS library should use only cores $t+1$ to 23. As GotoBLAS implements its own thread pinning policy a patch was implemented that allows for excluding CPU cores from the GotoBLAS pinning routines.[13] Fig. 6 shows the performance achieved by the first implementation.

2.4 DGEMM Optimizations

CALDGEMM was optimized in two steps. First the kernel performance was maximized, as it does not depend on the framework at all. Second the framework was improved using the fastest possible kernel.

2.4.1 Kernel Optimization

This section will describe what was done to improve the kernel. Older GPU DGEMM implementations, especially for NVIDIA GPUs, usually used Shared Memory / LDS as described

[11] Each Magny-Cours CPU internally consists of two dies with one memory controller each.

[12] t will be introduced later and can currently considered to be 0.

[13] The GotoBLAS library implements two routines for this: one respecting NUMA and one without NUMA. Although only the NUMA variant is used on the cluster, both routines were patched to allow for the greatest possible compatibility.

in [Vol+] for example. The LDS, however, is not capable of delivering sufficient bandwidth to get full ALU utilization on Cypress GPUs. Therefore, it is better to read the input data through the texture cache. This is analyzed already in [Nak].[14] This section will only handle the processing of one tile $C_{i,j}$ which will be called C hereafter. The tile will be subdivided into $C_{i,j}$ again, which is, however, not related to the tiling above. This also holds for A and B. The reader should keep this in mind so that no ambiguity occurs.

To achieve maximal performance the CALDGEMM kernel was written in the AMD Intermediate Assembler Language (IL) [Amd2]. To minimize the amount of data required from the memory a technique called blocking or also tiling is implemented. As tiling has already been used in another context throughout this document it will be called blocking hereafter.

Blocking To calculate the DGEMM $m \cdot n \cdot (2k + 2)$ floating point operations are needed. Reading the input data from memory independently requires $m \cdot n \cdot 2k$ memory fetches. This amount of fetches is reduced by the blocking. The matrix C is subdivided into blocks $C_{i,j}$ of dimension $a \times b$ where a is the vertical blocking size and b the horizontal. The matrices A and B are also subdivided into $A_{i,j}$ and $B_{i,j}$ of dimension $a \times 1$ and $1 \times b$ respectively. C can then be calculated by $C_{i,j} = \sum_{l=1}^{k} A_{i,l} B_{l,j}$. To do this, in each step the matrices $A_{i,l}$ and $B_{l,j}$ are loaded into registers, and each element of the one is multiplied with each element of the other and then added to the corresponding entry in $C_{i,j}$ (See Fig. 7).

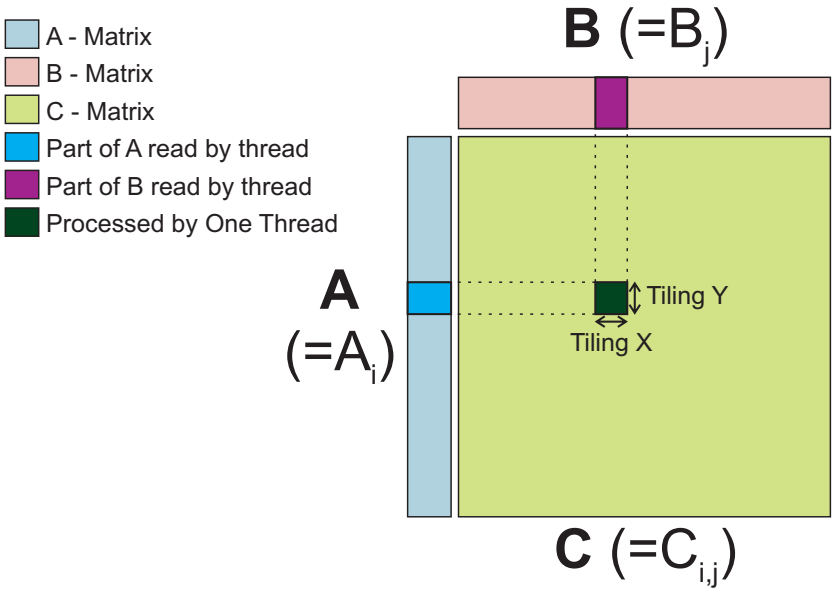

Figure 7: Tiling / Blocking of DGEMM Kernel

This requires only $k \cdot (a + b)$ memory loads to calculate $a \cdot b$ entries of C, so in total

$$\frac{m}{a} \cdot \frac{n}{b} \cdot k \cdot (a + b) = mnk \cdot \frac{a + b}{ab}$$

memory fetches. The amount of fetches is thus reduced by a factor $\frac{2ab}{a+b}$.

The peak performance in double precision of the GPUs employed is 544 GFlop/s. Using a 4×4 blocking the available L1 cache bandwidth provides exactly the throughput available to

[14] [Nak] also provides a faster kernel implementation than the AMD Stream SDK, but this was not available when the CALDGEMM work started.

achieve 544 GFlop/s, given that cache hit efficiency is 100%. This is of course unrealistic and suggests a blocking slightly bigger than 4×4.

The following blocking variants were implemented: 8×2, 4×4, 8×4, 4×8, and 8×8.

Register Usage The blocking is used in the way, that each GPU thread calculates one of the $C_{i,j}$ submatrices of C. Therefore, the thread needs at least $a \cdot b$ registers for $C_{i,j}$, a registers for $A_{i,l}$ and b registers for $B_{l,j}$ plus 3 registers for i, j, and l making a total of 3 integer registers and $(a + b) \cdot (b + 1) - 1$ double precision registers. As the GPU offers 128-bit registers only $\frac{(a+1)\cdot(b+1)+1}{2}$ registers are needed. All kernels benchmarked later were implemented such that they really hit this lower register boundary, which is e.g. 41 for the 8×8 kernel, the biggest one implemented. A simple trick to reduce the amount of registers is to use multiple input data buffers. This way the same relative pointer can be used in multiple buffers. Obviously, to hide memory latencies, the highest possible number of threads should be running concurrently. However, for the limited number of registers, thread-count and blocking size cannot both be maximized at the same time. This shows that a larger blocking is not always preferable and that the trade-off point has to be found.

In the end it turned out, that for the 4×4 kernel the register usage is not that critical. For the unrolled kernels the compiler assigns more registers. Nonetheless, they are still faster.

Unrolling & Hardcoding Constants In the kernels, multiplying $A_{i,l}$ with $B_{l,j}$ is done explicitly, with a single loop over l. This loop can be unrolled to achieve optimal performance. Fig. 8 shows the performance of an exemplary kernel using different unroll factors. The optimal unroll factor depends on the blocking size. (Bigger blocking sizes need less unrolling.) For each kernel the optimal unrolling factor was experimentally determined individually.

Further the k constant can be hardcoded in the kernel instead of loading it from a constant buffer. For the HPL $k = nb = 1024$ was used. To account for this a special kernel with hardcoded $k = 1024$ is added. At runtime the correct kernel is called depending on the actual k. Fig. 9 shows the performance benefit using hardcoded k instead of loading the same k from a constant buffer. As can be seen a hardcoded k does not always improve kernel performance. However, for the B transposed kernel variant, which was used for the HPL in the end, the performance increased.

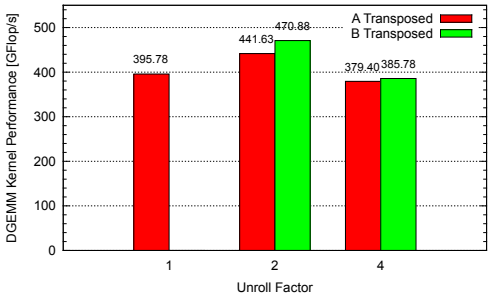

Figure 8: Performance of unrolled DGEMM Kernels

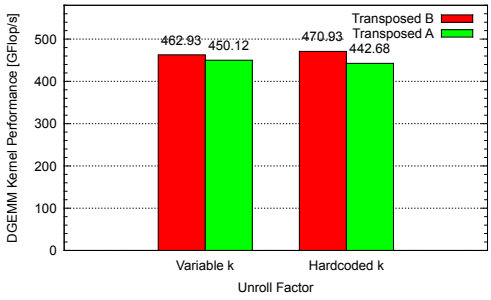

Figure 9: Performance using hardcoded k

Cache organization The texture cache can be configured in two ways: linear or tiled[15] where the tiled mode is optimized for two-dimensional access patterns. Accessing the matrices

[15] See [Amd1] for a description of the tiled mode.

for the DGEMM is a two-dimensional pattern and the tiled mode is by far the faster mode for CALDGEMM as can be seen in Fig. 10.[16]

Output Memory / Shader Type The Cypress chip offers two different possibilities to write data to the output memory:

- **Color Buffers:** In original pixel shaders Color Buffers were the only possible output type. Each thread can output to at max eight buffers of float4 entries. The output position in the buffer is determined by the position of the thread in the execution configuration, so no address calculation is necessary. However, as a thread calculates adjacent entries of the destination matrix but the output is written to eight distinct buffers, the output format cannot be chosen to be the native memory format of the C matrix. This already shows that the MergeBuffer function is mandatory when using Color Buffers. Additionally, eight float4 color buffers allow for storing 16 double values and are therefore unsuited for larger blockings.[17]
- **MemExport:** The newer AMD chips as the Cypress allow for the MemExport function. This way threads can directly address and access a linear destination buffer (The Global Buffer). However, only one Global Buffer can be used at a time, which leads to some problems as discussed in section 2.5. All kernels implemented with large blocking use the MemExport function in lack of alternatives. Additionally, the smaller kernels have been implemented using the MemExport function, too.

Besides the output method different output locations can be chosen. Either the output can be written to the global GPU memory or to page locked host memory. In the first case the data is transferred to the host after the kernel execution in one big DMA transfer. In the second case the output is directly written to the host memory via DMA by the kernel. So either an extra DMA transfer is required or the kernel execution might be slowed down due to host memory access. For each kernel variant the better implementation is experimentally chosen.

The performance of the output types is shown in Fig. 14 in the overall kernel comparison at the end of this section.

In addition to the common Pixel Shader the Cypress chip also supports Compute Shaders. Compute Shaders are meant not for graphics but to allow for more complex GPGPU programs, e.g. they only support the MemExport output. It turns out that the Compute Shader version is inferior to the Pixel Shader version with MemExport, although it is identical except for the Shader Type definition. The Compute Shader approach was thus not followed.

Matrix Size The matrix size affects the cache access pattern and is thus relevant for the performance. As the original matrix for the DGEMM is tiled anyway into submatrices processed by the kernel, the kernel matrix size can be chosen almost arbitrarily in a way that maximizes the performance. As the dependency on m and n is alike in the following $m = n$ is assumed. Further, according to section 2.1 the matrix can be assumed a square matrix. Thus in the complete section on the DGEMM kernel it is assumed that $h = m = n$.

[16] It has to be admitted that the kernel used for the comparison has been tuned for the tiled mode. I should be possible to improve the kernel for linear mode quite a bit. However, tuning for linear access is much more complicated, and it is not sure whether the same performance - close to peak performance - can be achieved.

[17] The 4×4 and 8×2 blocking can use Color Buffers. All other blockings are restricted to MemExport.

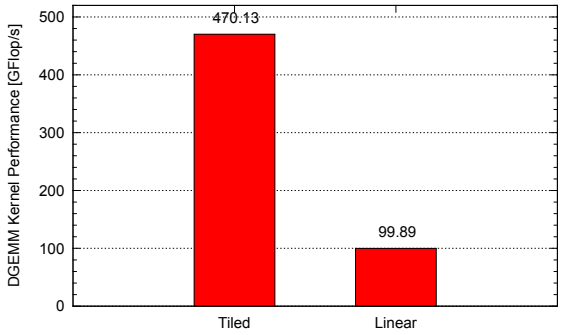

Figure 10: Performance for different cache types

Fig. 11 shows the performance for multiple h and k values. The kernel used here is the best-performing kernel that is determined at the end of this chapter. Showing all plots for all kernels would by far exceed the scope of this document. Using the plots, for each kernel the matrix sizes suited best were determined.

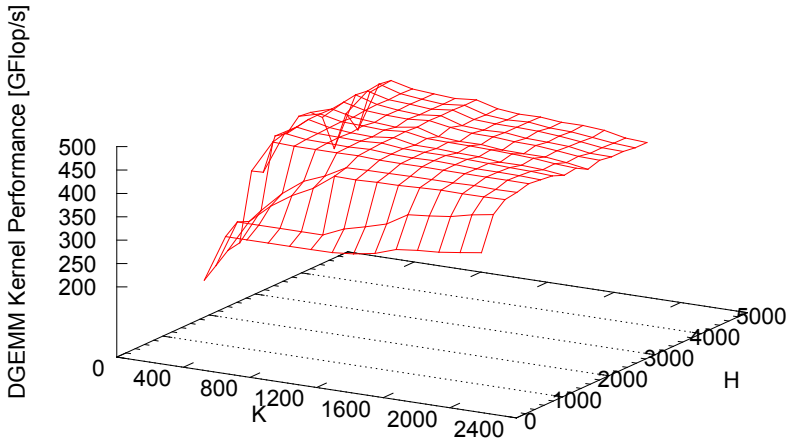

Figure 11: DGEMM Kernel Performance for different Matrix Sizes

Fig.11 shows that performance decreases significantly, when m drops below 1024, it very slightly increases above 1024. At $k = 1024$ is has a peak, that is related to the special treatment of $k = 1024$ described above. Above 1024 it slightly decreases. At about $k = 512$ it raises again with its total peak at $k = 448$. However, using such low k introduces more overhead for the synchronization etc. Therefore, even as the kernel is not the absolute fastest one, $k = 1024$ remained the chosen value for the HPL. Fig. 12 and 13 show more detailed plots for some chosen k and h.

The final values used for h are 512, 1024, 2048, 3072, and 4096 depending on the parameter size. Section 2.5.1 gives more information about this. The value 512 is used only when absolutely necessary, namely in the case when $m < 1024$ or $n < 1024$. (Be aware that m and n are the total matrix dimensions here and not the tiling size).

Transposed Matrices The DGEMM specification itself allows for transposed input matrices. Therefore, either the kernel or the DivideBuffer routine must be capable of transposing

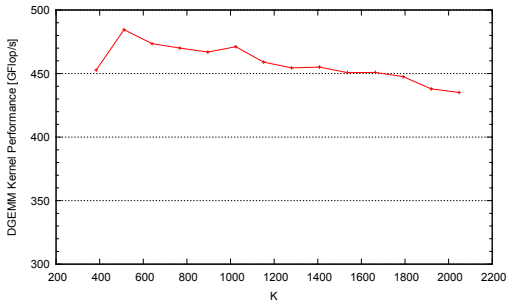

Figure 12: Kernel Performance at different k

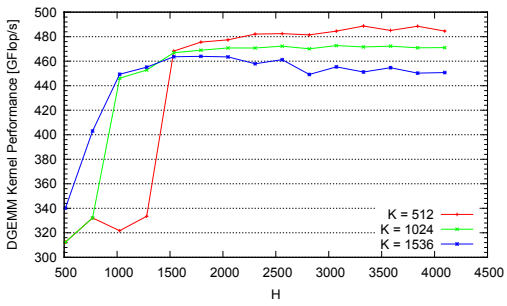

Figure 13: Kernel Performance at different h

input matrices. However, as already said, the treatment will be described later. At the moment only the kernel performance shall be considered.

The GPU always reads a double2 value at a time. For an untransposed A matrix this means that two columns are read at once. This is unsuited regarding the blocking approach, where only one column is required at the same time. To avoid this, the data structure of the input matrices can be altered, such that two consecutive doubles in memory correspond to the same column. The conversion is done by DivideBuffer. Doing this, kernels without transposition, A transposed, B transposed, as well as both A and B transposed were implemented with all blocking methods. For symmetric blocking the case with both matrices transposed is identical to the one with none transposed, so it was omitted. For the large 8×8 tiling the B transposed part was omitted, as the A transposed version turned out not to perform well and the effort to introduce the special data structure was considered inappropriate.

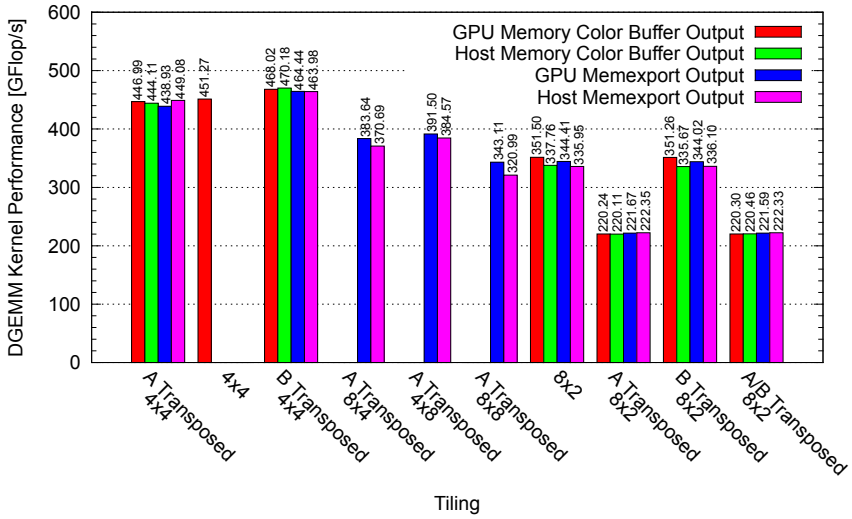

Figure 14: DGEMM Kernel Performance Overview

Fig. 14 shows the results using all kernels with all transposition, blocking, and output variants. All results presented are tuned individually using the best unrolling factor and matrix tiling sizes.

It is obvious that the 4×4 kernel with transposed B matrix is the fastest one. It turns out that using the MemExport function is generally a bit slower than using Color Buffers. This is good news as the Color Buffer output offers multiple buffers whereas MemExport is restricted to a single output causing a problem as described in section 2.5.

Comparing the output location unveils an interesting fact. For most kernels using the GPU memory for output is faster, as it is expected. However, for the 4×4 kernel with transposed B it is the other way around. Most probably reading the input data exhausts the global GPU memory bandwidth but data can be written to the host memory independently. This fact is very welcome, as writing to host memory directly is also the best case from the synchronization perspective.

In summary the kernel used in all later benchmarks has the following characteristics:

- 4×4 blocking.
- B matrix transposed.
- Pixel Shader kernel.
- Output done using Color Buffers.
- Output buffer is located in host memory and accessed by the kernel directly via DMA.
- Loop unrolled with an unrolling factor of two.
- $k = 1024$.
- $h \in \{512, 1024, 2048, 3072, 4096\}$ (chosen by runtime as described in section 2.5.1).
- Texture cache is used in tiled mode.

2.4.2 Scheduling

As for all asynchronous processes scheduling is required in CALDGEMM. The final goal is to utilize both the GPU and CPU to the full extent and hide all latencies. Further it is desirable not to split the matrix into too small tiles as both CPU and GPU DGEMM work better on larger matrices.

GPU/CPU Performance Ratio As described in section 2.3 the matrix is split in two parts where one part is processed by the GPU and one by the CPU. It shall now be explained how the best splitting ratio between the two components is determined. Obviously the splitting ratio is not a constant but can depend on the input parameters m, n, and k. A plausible assumption is that it does depend on the size of C but not on the shape, so it depends on $m \cdot n$ but not on m and n individually.

Fig. 15 visualizes the ratio of GPU and CPU performance as a function of $m \cdot n$ using an assortment of runs with different m and n. Optimally the plot should show a curve that goes asymptotic towards a constant for large $m \cdot n$. Unfortunately, this is not the case. A deeper analysis unveils that this is related to a slowdown in GotoBLAS for certain input parameters. The green points in the figure correspond to runs with $n = 2048$, red points to runs with $n \equiv 0 \pmod{4096}$ and blue points to all other parameters.

To understand the behavior at $n \equiv 0 \pmod{4096}$ Fig. 16 shows the performance in a range of 2048 around $n = 40960$ (which is $\equiv 0 \pmod{4096}$). It clearly demonstrates that the slowdown only occurs in a small area around $n = 40960$.

Fig. 17 shows the region near $n = 40960$ in a higher resolution. To exclude the possibility that the slowdown is related to the thread count two variants are shown; with similar behavior.

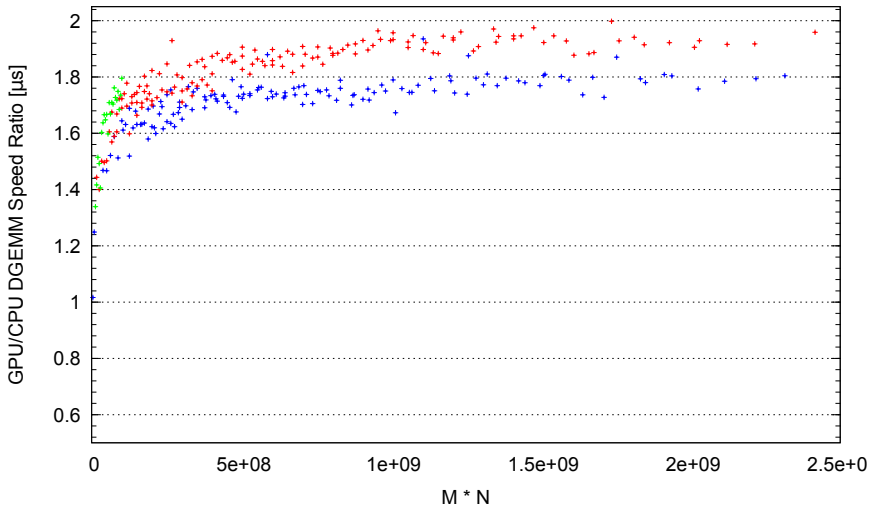

Figure 15: GPU / CPU DGEMM Performance Ratio

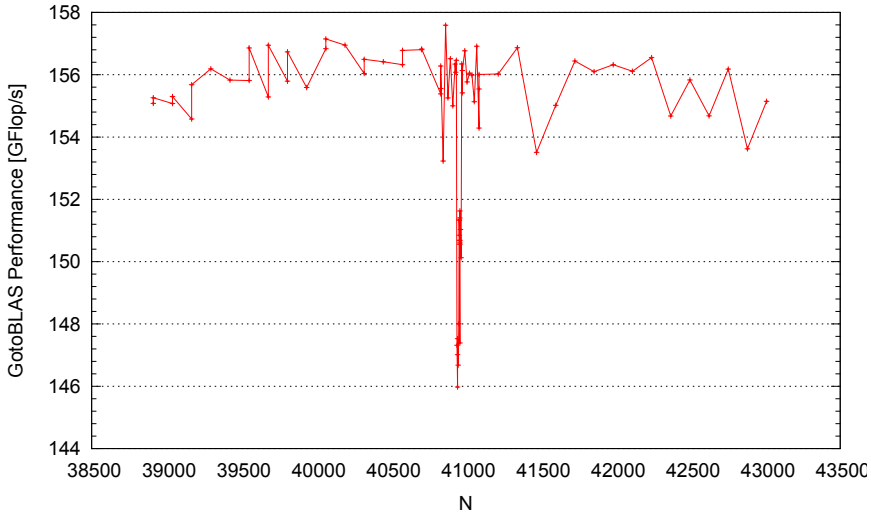

Figure 16: GotoBLAS DGEMM Performance using 21 Threads depending on n

Fig. 18 shows that there is no such dependence on the m parameter. As a conclusion there are two special cases, $n = 2048$ and $n \equiv 0 \pmod{4096}$ that have to be treated independently, otherwise it can be assumed that the ratio depends only on $m \cdot n$.

Fig. 19 shows two fits to the experimental ratio results. The problematic input parameters are excluded in this case. In the data for the separated GPU / CPU ratio, independent DGEMMs were executed on GPU and CPU respectively. However, finally they should run concurrently and are likely to influence each other. Therefore, a new data-set was created, executing a combined GPU/CPU DGEMM and measuring the GPU and CPU performance contribution to the combined DGEMM independently. In this second combined run the scheduling was based on the data obtained by the previous separated runs. A new fit was done for the new data-set. The resulting curve is used to determine the ratio for CALDGEMM.

The two cases excluded for the fit are handled the following way: If n is detected to lie in the problematic region, the ratio is at first determined ignoring the GotoBLAS slowdown. To account for the slowdown it is then corrected as for a 5% decreased CPU performance.

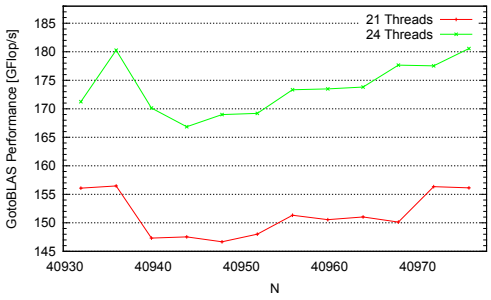

Figure 17: GotoBLAS Performance near $n = 40960$

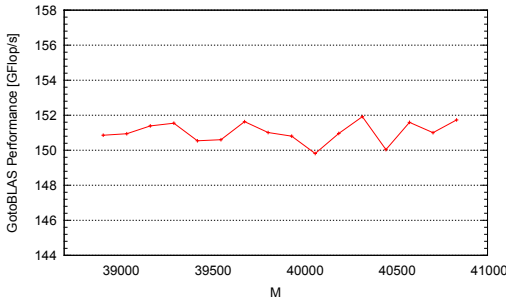

Figure 18: GotoBLAS Performance depending on m

This follows the principle introduced in section 2.3 to rather overestimate the GPU performance. The small inaccuracy introduced by this is then treated by the advanced scheduling mechanisms described in the next section.

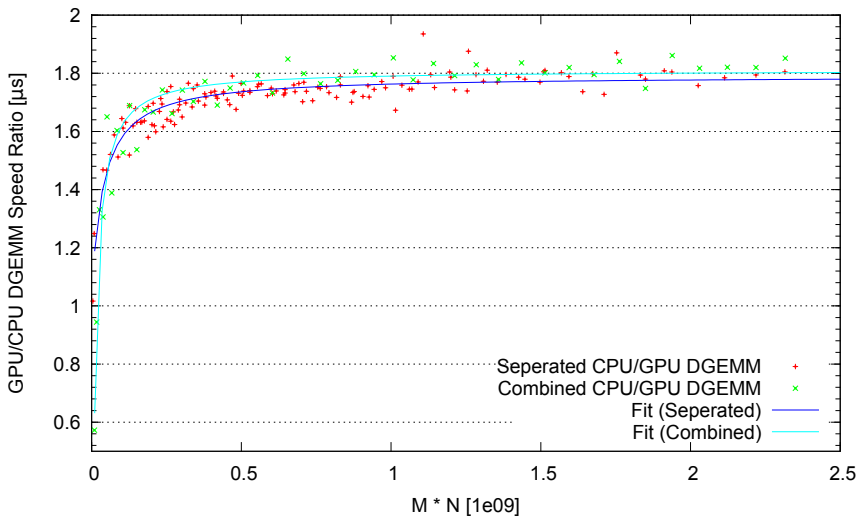

Figure 19: Fitted GPU / CPU DGEMM Performance Ratio

One additional fact must be considered. The CPU also has to process the rightmost part of the matrix, as long as n mod $h \neq 0$. The size of the overlapping region depends on m and n. This is handled the following way. First the CPU part is calculated but the splitting position is not yet adjusted to a multiple of h. It is then downsized to correct for the overlapping region and only then adjusted to be a multiple of h.

2.4.3 2nd & 3rd Phase

Clearly using only the static scheduling introduced yet will not result in optimal performance as the duration of a DGEMM call changes randomly. Additionally, to ensure the correct size of the GPU part of the matrix, the splitting position cannot be chosen arbitrarily but only in steps of h which is 4096 for large matrices. Therefore, a more fine granular scheduling is needed. It could be argued that GotoBLAS could process the same tiles used for the GPU. However, Fig. 20 shows that even the largest 4096×4096 tiles are insufficient to achieve optimal GotoBLAS performance. Therefore, it is desirable to schedule the largest possible submatrices.

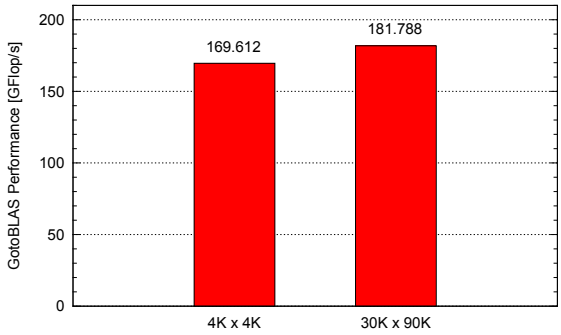

Figure 20: GotoBLAS performance dependency on matrix size

To account for this a second and third phase was introduced in the scheduler (The first two CPU DGEMMs already explained are both considered phase one). As just explained, the splitting position cannot be chosen arbitrarily. The correct splitting position, however, usually lies between two possible positions, which are h rows away from each other. When the CPU has finished the first phase run it checks how many GPU tiles are still unprocessed. It uses the same ratio as for the first splitting to determine how many GPU tiles it should process. It then takes a rectangular section containing this amount of tiles. This is called second phase run.

With the CPU having finished its second phase run the GPU is also likely to finish soon. To process a tile the CPU requires about three times the time the GPU does. However, after the GPU has finished its last tile the output must still be postprocessed by MergeBuffer. This takes about as long as the kernel execution. Additionally, in average the GPU will still have to process 50% of its current tile. This leads to the following rule for third phase runs. As long as there are still at least two unprocessed tiles the CPU will "steal" one from the GPU. All these runs on stolen tiles are called the third phase. Fig. 21 shows an example matrix distributed between the GPU and different CPU phases. (The second phase is not necessarily restricted to one row of tiles.)

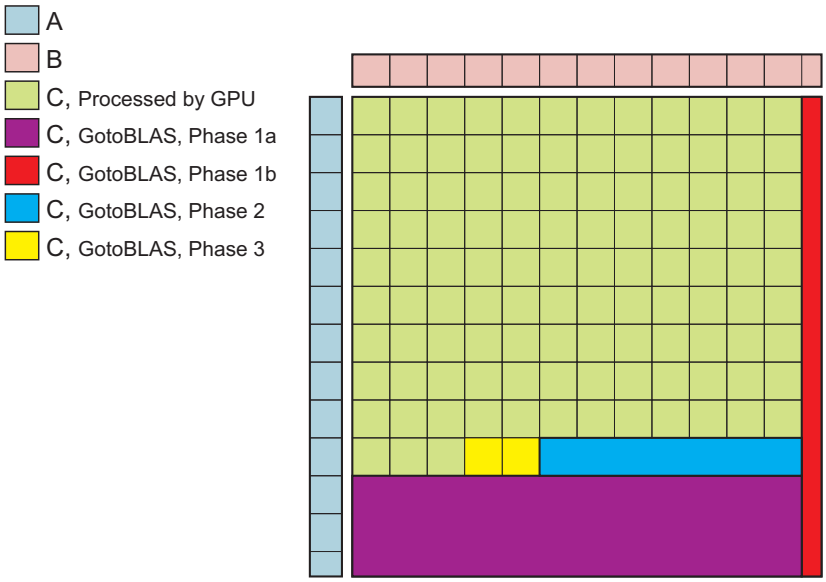

Figure 21: GPU / CPU Distribution of C Matrix

2.4.4 Transfer Optimizations

Asynchronous Transfer Up until now pre- and postprocessing, DMA transfer to the GPU, and kernel execution were completely serialized. Only the DMA transfer to the host is already done in parallel to the computation as described in chapter 2.4.1 and thus completely hidden. It is desired to parallelize these tasks as far as possible. The solution is to introduce a pipeline. In the n-th iteration the CPU can preprocess tile n. In the meantime tile $n-1$ can be transferred to the GPU via DMA. Concurrently the GPU can already process tile $n-2$. And again at the same time another CPU thread can postprocess tile $n-3$. Using this to the full extent only leaves the pure GPU calculation time as runtime together with some overhead. This overhead consists of the accumulated time for DivideBuffer, DMA transfer to GPU, and MergeBuffer; however, only for a single tile.

With the current drivers, a DMA transfer cannot be started while a kernel is being executed. However, a DMA transfer that is already running is not affected by kernel execution. Therefore, in the pipeline the CPU must be two steps ahead of the GPU. Before the CPU calls the kernel for tile n it must issue the DMA transfer for tile $n+1$. The overhead consists of the time for DivideBuffer, MergeBuffer, and twice the DMA transfer. In the implementation, however, the CPU first prepares only one tile, transfers it to the GPU and calls the first kernel. While the first kernel is running the CPU preprocesses tile two and three. However, tile two cannot already be transferred while the first kernel is running. This is the point where the overhead of the second DMA transfer occurs. Anyway the alternative would be to prepare two tiles in the very beginning. This is not wanted as DMA transfer time is lower than the DivideBuffer time (See Fig. 24).

As the MergeBuffer routine runtime exceeded the kernel execution time, or was at least of the same order of magnitude, to ensure a continuous kernel execution two MergeBuffer threads are executed. The threads postprocess the output alternately. Now let t be the number of output threads. Together with one thread for DivideBuffer this makes $t+1$ threads to handle the GPU DGEMM.[18] As, in addition, the GPU writes directly to one output buffer in host memory in total $t+1$ output buffers are needed. These are used in a cyclic way.

BBuffers Recapitulating the way the tiles are streamed to the GPU, it becomes obvious that not all matrices A_i and B_j need to be transferred at the same time. The matrix A_i is transferred to process $C_{i,1}$. The following tiles in the stream, $C_{i,2}$ to $C_{i,n/j}$ also require A_i which thus should not be retransferred but simply remain on the GPU. Afterwards A_i is never used again and A_{i+1} is transferred. Now consider the B matrix: when calculating $C_{1,1}$ to $C_{1,n/h}$ each B_j is used exactly once. As long as it is possible to store all the B_j matrices in GPU memory, they never need to be retransferred afterwards. The buffers for the B matrices are called BBuffers. The current implementation tries to allocate 21 BBuffers.

For non square C matrices the matrices A and B require a different amount of memory. Swapping the inner and the outer loop of the streaming process swaps the roles of A and B in the above consideration. So the buffer in the GPU memory only needs to store the smaller one. Therefore, currently the buffer is small enough as long as the C matrix' smaller dimension is below $21 * 4096$. It has to be noted, that the CPU also processes a part of the matrix. This can be chosen in a way to lower the smaller dimension if it would exceed the buffer size otherwise. Finally, to exceed the buffer size, C would have to be at least of dimension 129024×129025 what would require 124 GB of main memory.

[18] This explains the t variable in section 2.3.1. The patch already handles the increased GPU thread counts. All these threads are pinned to core 0.

Altogether, the BBuffers ensure, that both the A and B matrix are transferred exactly once.

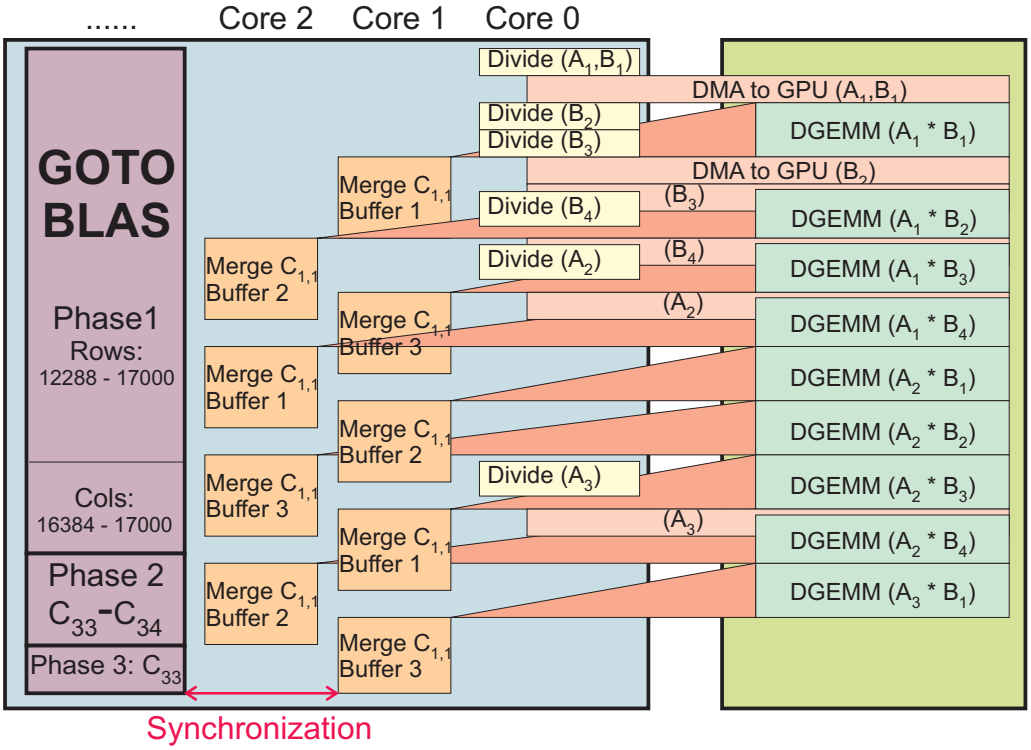

Figure 22: Process-Flow of new CALDGEMM Implementation

Fig. 22 visualizes the implementation while Fig. 23 show the effect of both optimizations individually and in combination. It can be seen that both attempts provide a large performance benefit. The combination, however, does not result in such a huge leap in performance because both optimizations approach the same inefficiency. Anyway, especially for smaller matrices, using the combined implementation delivers by far the best performance.

DivideBuffer & DMA As the GPU kernel can write its result back directly to host memory also the DivideBuffer routine could write its output directly to the GPU via DMA. This would make the subsequent large DMA transfer unnecessary. However, Fig. 24 shows that a DivideBuffer writing directly to GPU memory takes more time than the usual DivideBuffer and the DMA transfer together. This approach was thus discarded.

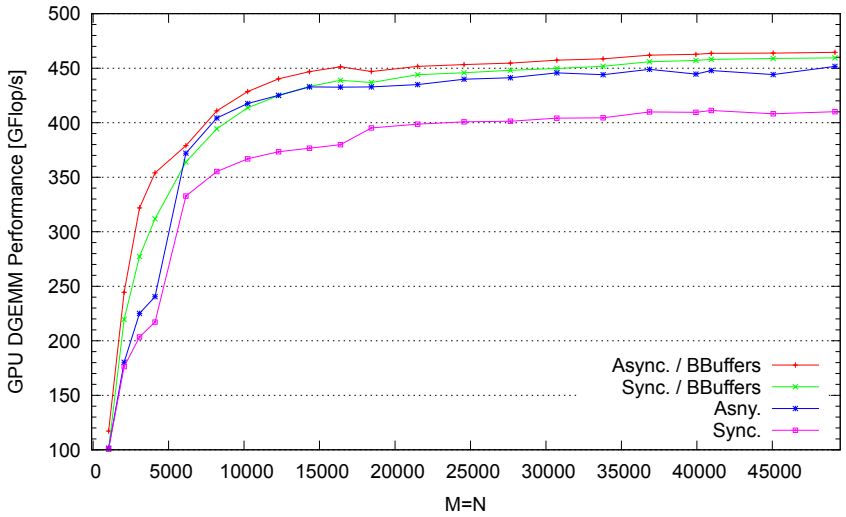

Figure 23: GPU DGEMM Performance for Asynchronous Transfer and BBuffers

2.5 Patched AMD Driver & Vectorization

Fig. 24 shows the throughput of the DMA transfer and the pre- and postprocessing routines.[19] Consider the DMA bandwidth at first. Benchmarks showed a huge dependance on the employed system BIOS version. With earlier versions the throughput was only about 3 GB/s. Still with the new version the transfer to the GPU is inferior to the transfer to the host. This is unfortunate, as only the transfer to the GPU is used in practice (as the kernel writes directly to the host). However, considering the matrix sizes appearing during the HPL and remembering that all matrices A, B, and C are transferred exactly once, the GPU transfer is not so relevant anymore. E.g. for $k = 1024$, $m = n = 81920$ the data transferred to the host is 40 times the data transferred to the GPU.

All versions of the DivideBuffer and MergeBuffer routine were vectorized.[20] Prefetches[21] are used and also streaming stores[22] are employed where they brought a benefit.

It becomes obvious, that the DivideBuffer routine is much slower when it has to transpose the input matrix. Section 2.5.1 will answer the question whether this is acceptable.

The results for MergeBuffer need some explanation. Consider only the results without the hacked driver first. It turns out, that the MergeBuffer routine is faster, when the data is transferred in one large DMA transfer than when the kernel writes directly to the host memory. The reason is the following: For the kernel to have access to the memory, the memory region must be unmapped, i.e. it has no virtual address. For MergeBuffer to access the data, the memory must be mapped again. This results in one page fault for every page accessed. This is very unfortunate as the data is only read once. In fact, executing a second MergeBuffer call on the same buffer, without un- and remapping yields in the same performance as with the single large DMA transfer.

[19] The DMA transfer to the host is measured using a different kernel writing to GPU memory, and doing the DMA transfer afterwards. Obviously, using the best-performing kernel writing direct to host memory, makes the DMA throughout impossible to measure.

[20] See Appendix of [Roh1].

[21] Memory is explicitly loaded into the cache before its actual use to omit memory latencies then.

[22] A streaming store bypasses the cache and write directly to the memory.

Thus, the problem is that the driver forbids kernel calls while the memory is mapped, while it does not forbid explicit DMA transfers. Patching the AMD driver and removing the check whether the buffer is mapped solved the slowdown. As this is definitely no nice solution, especially for a release of the library, this will be referred to as the "Driver Hack" hereafter.[23]

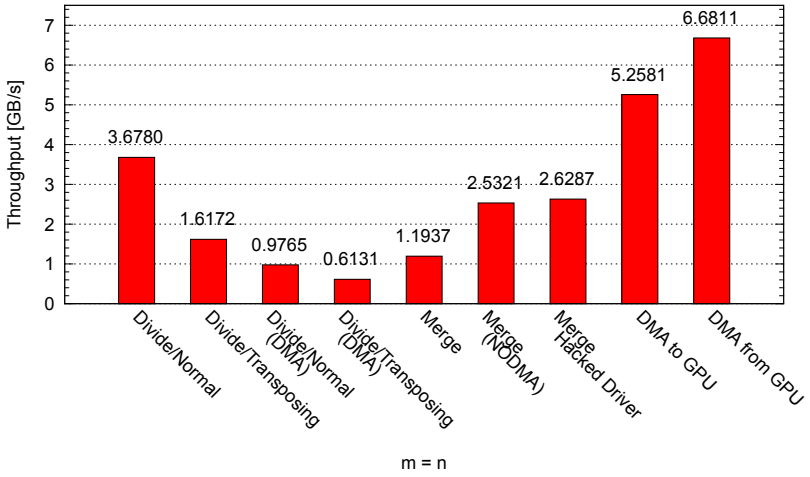

Figure 24: Performance of Pre- / Postprocessing and DMA Transfer[24]

Global Buffer Summary Recall that a global buffer is required for the MemExport function that is currently not used in CALDGEMM. Therefore, there seems not much sense in discussing it here. However, the current kernel with 4×4 tiling is coming to its very limit. For any further performance increase a bigger blocking is required. But then MemExport must be used and thus also the global buffer.

The problem is: only a single global buffer is usable at a time. Currently, when using t output threads CALDGEMM requires $t+1$ output buffers. As only a single global buffer is available, these output buffers must all reside in this single big buffer. But then a problem arises when running a kernel and MergeBuffer simultaneously, as MergeBuffer requires the buffer to be mapped while a kernel without the hack would not start when the buffer is mapped. This makes the driver hack mandatory when using the global buffer. With the driver hack, the global buffer can be used in CALDGEMM, only the performance is slightly decreased.

2.5.1 Miscellaneous Optimizations

Tiling Size In section 2.4.1 the effect of the matrix size on kernel performance was analyzed. It was concluded, that $h = 512$ can be used for matrices with $n < 1024$ or $m < 1024$. Otherwise, all 1024, 2048, 3072 and 4096 bring good results. However, up until now only kernel performance was taken into account. Now the effect on overall performance shall be discussed. Clearly, on the one hand, a larger h reduces the synchronization overhead. On the other hand, the overhead before the first and after the last kernel call increases. However, this overhead for large h is constant, so its relative effect should decrease for large m and n.

[23] AMD is aware of the problem, and for OpenCL kernels the corresponding check is already removed in the driver.

[24] In results for DivideBuffer marked as DMA the routine was writing directly to GPU memory instead of CPU memory. The MergeBuffer result marked as NODMA is a run where the kernel wrote its output to GPU memory with one big DMA transfer afterwards.

Therefore, the best tiling size is expected to depend on m and n. Again, it can be shown that it depends only on $m \cdot n$ so again $m = n$ is assumed. Fig. 25 shows the performance for different m.

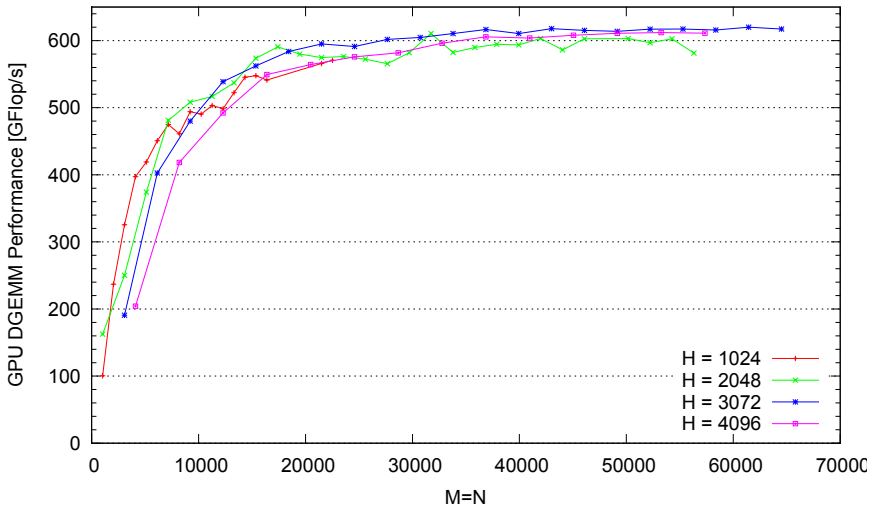

Figure 25: Performance for different CALDGEMM Tiling Sizes

It can be seen that as expected the optimal tiling size depends on the matrix size where bigger matrices favor bigger tilings. The tiling size is thus auto-adjusted to be optimal for each input matrix.

One issue with the automatic tiling size selection shall be discussed. It is unsupported by the API to transfer only a part of the buffer via DMA to the GPU. As the allocation takes comparably long the buffers are created when the library initializes. This predetermines the buffer sizes. As the BBuffers multiply the amount of input buffers it is not possible to allocate distinct buffers for all tiling sizes. Instead only a buffer for the maximum tiling size is allocated. For smaller h parts of these buffers are used. This clearly is a huge overhead in the DMA transfer, especially for a small tiling. However, there is no other possibility to solve the issue and benchmarks show that performance increases regardless of this inefficiency.

Transposed Matrices Up to now the correct input type was not considered. Both the possibility to use transposed matrices in the kernel and to transpose the matrix during DivideBuffer was discussed. It was concluded that for the kernel a transposed B matrix was optimal. In contrast, consider a transposed A matrix. Using the transposed B kernel for the transposed A input both matrixes have to be transposed in the DivideBuffer function, which is the worst case for the DivideBuffer performance. Alternatively the kernel for the transposed A matrix could be used resulting in the best case for DivideBuffer. Fig. 26 compares the 4 possible combinations of kernel and input type.

It turns out that the transposed B kernel is faster in both cases. In the end even the transposed A matrix is faster using the transposed B kernel. The explanation lies in the GotoBLAS library which is slightly faster when A is transposed. Therefore, the kernel for B transposed is used for all cases, even when only the A input matrix is transposed.

However, there is still one situation imaginable, where the transposed A kernel could be an interesting alternative. For smaller matrices the DivideBuffer function consumes a more significant amount of time. So here the overhead for the transposition weights more and the transposed A kernel could be faster. The problem is that the data arrangement of the buffer

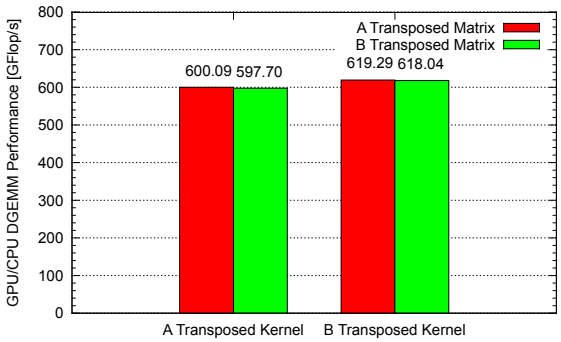

Figure 26: Performance for transposed input Matrices

must be declared at allocation time. As, using the BBuffers, much memory is needed for the buffers it seems inappropriate to allocate all buffers twice, for the transposed and non transposed case. However, as the transposed A kernel, or the kernel without transposition, would be used for small matrices only anyway, the BBuffers could be omitted (or reduced) in that case. Only one additional buffer (or only few) could be allocated for every case and the library could choose at runtime which kernel to execute. This feature has not yet been implemented but is planned for a future release.

2.6 Summary & Results

In summary CALDGEMM is implemented the following way:

- The matrix size handled by the GPU can easily be subject to restrictions with the CPU handling the overlapping parts.
- The transposed B kernel is always used regardless which matrix must be transposed or not.
- The optimal tiling size is chosen automatically depending on the input matrix size. Possible values are 512, 1024, 2048, 3072, and 4096.
- The splitting ratio between GPU and CPU is automatically chosen depending on the input matrix size.
- 2nd and 3rd phase GotoBLAS runs ensure that both GPU and CPU are fully utilized and do not idle.
- Using a pipeline, asynchronous transfer, and buffers on the GPU can make the GPU execute DGEMM kernels continuously.
- A patch needed to be applied to the AMD driver to achieve full performance.

Finally CALDGEMM can deliver a pure DGEMM performance of 620 GFlop/s on our test setup. The DGEMM performance does not depend on whether matrices are transposed or not. It slightly depends on the input parameters but stays high as long as the matrices do not get too small. Fig. 27 gives an overview of the final CALDGEMM performance.

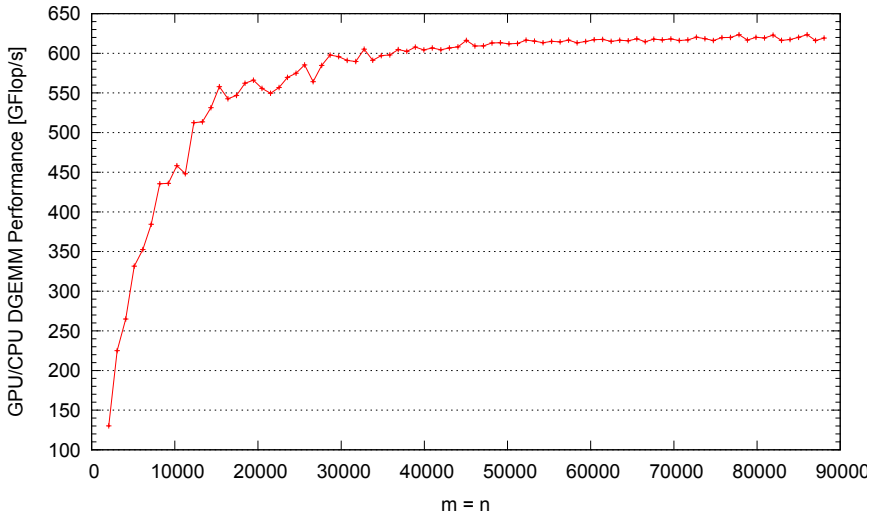

Figure 27: CALDGEMM Performance Overview

3 GPU based HPL

3.1 Integrating CALDGEMM

In the previous sections a fast DGEMM implementation was developed. This will now be integrated in the HPL. CALDGEMM only provides DGEMM but no full BLAS capability and it is based on GotoBLAS itself. Therefore, a combination of HPL, CALDGEMM, and GotoBLAS is used. The question is how to integrate CALDGEMM. As CALDGEMM is itself multithreaded and already involves multithreaded GotoBLAS calls, the common approach with multiple MPI processes per node, one per core, is not suited. Lots of problems would arise, e.g. which process would use the GPU, how to handle processes with different DGEMM performances, etc..

The way to go is a single MPI process per node, which is itself multithreaded. For the HPL pthreads and TBB[25] were used. As a result, other tasks apart from DGEMM had to be parallelized. The LASWPs, DLATCPY and DLACPY were parallelized and vectorized.

Some initial benchmarks revealed that $k = 1024$ is an appropriate value for the GPU. Lowering k will reduce the GPU DGEMM speed whereas the previous analysis showed that increasing k further makes no sense.

Throughout an HPL run there are lots of DGEMM calls in the factorization and one huge DGEMM call outside the factorization. First some statistics about the DGEMM parameters and the time distribution of the DGEMMs were collected.

Fig. 28 shows that the large DGEMMs with big m and n are called exclusively with $k = 1024$. As expected Fig. 29 shows that DGEMMs with a significant distribution to the overall time are all executed with $k = 1024$. It was thus decided to use CALDGEMM only for the one single DGEMM outside the factorization step. In later attempts all other tasks of the HPL are hidden as far as possible to keep the GPU executing DGEMM kernels all the time.

The first GPU HPL version is implemented as shown in Fig. 30. The factorization and the pivoting is multithreaded. The DGEMM is multithreaded as well and in addition using the

[25] Intel Threading Building Blocks [Int2].

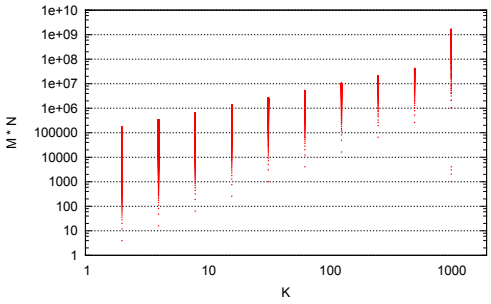

Figure 28: DGEMM Parameter Statistics during HPL Run (16 Nodes)

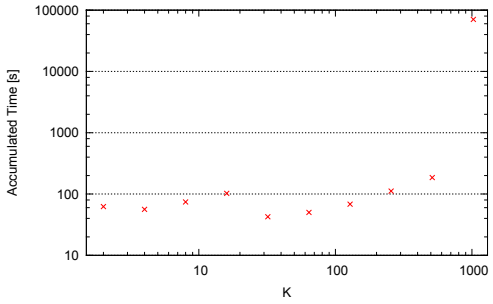

Figure 29: Statistics for DGEMM K-Parameter during HPL Run

GPU. Currently, the broadcast of the panel and the U-matrix is serialized and obviously not multithreaded.

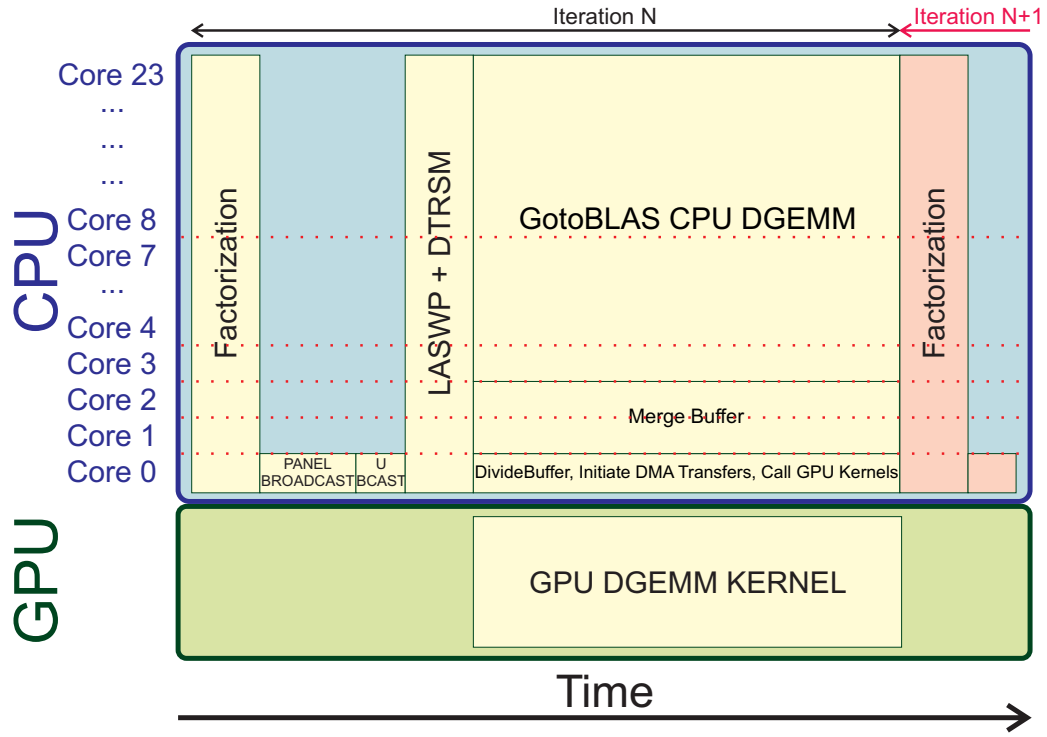

Figure 30: Process-Flow of GPU-based HPL

3.2 Optimizing HPL

The first implementation naturally leaves much room for improvement. At first the given HPL parameters are tuned to its best. Most parameters affect the network or the recursive factorization. For the Factorization the best parameters turned out to be (See [Hpl]):

- NBMin = 64
- NBDiv = 2
- Panel factorization: Crout oriented

- Recursive factorization: Left oriented

3.2.1 Alignment

Another optimization applied was a correction to the alignment. HPL usually aligns data to eight doubles, which already aligns the matrices to cache line size. However, multiple rows (or columns in col-major) are likely to have the same cache tag, so when accessing a single column only a fraction of the cache is used. The stride between the rows was thus altered such that the full cache is used. This change in the leading dimension of A in particular speeded up the LASWP processes.

These alignment corrections are also applied to the U-matrix, that is stored externally in multi-node runs.

3.3 Multi-Node HPL

With CALDGEMM integrated in HPL multi-node runs are immediately possible, however, lots of tuning can be applied. Again the available HPL parameters are tuned first. Fig. 31 shows that the long transfer mode is superior to the ring transfer mode (See [Hpl]). The long mode is usually used for fast nodes with a (relatively) slow network. In the LOEWE-CSC both the nodes and the network are very fast so it was not immediately clear which algorithm to use. Finally, for the cluster the modified long mode even performs better when using an increased amount of nodes. It also shows, that the original lookahead with depth one as implemented in the HPL has a negative effect. (Still, with the lookahead the performance of the ring and long transfer mode is equal. This shows that the lookahead successfully hides the communication time.)

As the cluster is equipped with QDR Infiniband the effect of RDMA was measured. Fig. 32 shows that using RDMA gives a performance increase.[26]

It turned out that copying the L-matrix for the transfer such that a continuous memory segment can be transferred brings a performance boost. It further turns out that transferring the entire L-matrix with the additional padding for correct alignment is faster than omitting the padding during the transfer.[27]

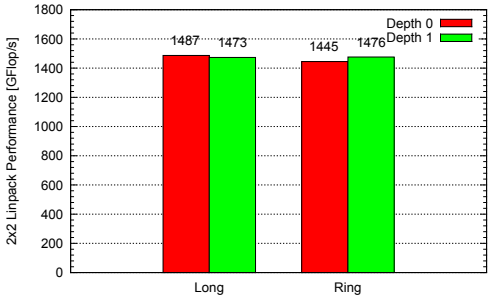

Figure 31: Influence of HPL Parameters on Performance

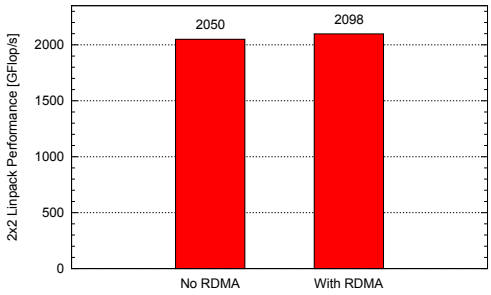

Figure 32: Performance Gain using Infiniband RDMA

[26] Unfortunately, currently there are unsolved problems that lead to errors when using PCIe DMA transfer to both, the GPU and IB at the same time. It only leads to instabilities when running with multiple hundreds of nodes. The following exemplary benchmarks were mostly done on four nodes with RDMA enabled. However, for large runs on the cluster RDMA has to be disabled for the time being.

[27] This clearly makes sense since without a padding no continuous segment is transferred any more.

3.4 Lookahead

As the original lookahead algorithm seemed useless a completely new lookahead was developed from scratch. The idea is to hide the factorization and the broadcast, and later even the swaps, to keep the GPU running DGEMMs at 100% of the time. The lookahead is implemented in multiple steps.

3.4.1 Lookahead 1

In a first lookahead step especially the panel broadcast time shall be hidden. (Panel broadcast takes much longer than U-broadcast.) As a matter of fact the panel broadcast requires the factorization to have finished. Therefore, to hide the panel broadcast also the panel factorization must be hidden.

Looking at the algorithm reveals, that the factorization only requires the DGEMM of the previous iteration to finish the first nb columns. Thus, it is possible to start with the next factorization while the DGEMM is still running. Having finished the factorization the panel can be broadcasted in parallel to the DGEMM computation, too. As the small column count of 1024 is unsuited for the GPU the first nb columns are excluded from the GPU DGEMM and handled by the CPU before the CPU even starts its large DGEMM. Between the two DGEMMs the CPU does the panel factorization and broadcast. This also makes the synchronization easier compared to the scenario where the GPU would process the first nb columns (with the streaming direction altered such that they are processed first).

A "Linpack-Mode" was introduced in CALDGEMM, where it supports callback functions for the factorization and broadcast. The first lookahead implementation is visualized in Fig. 33.

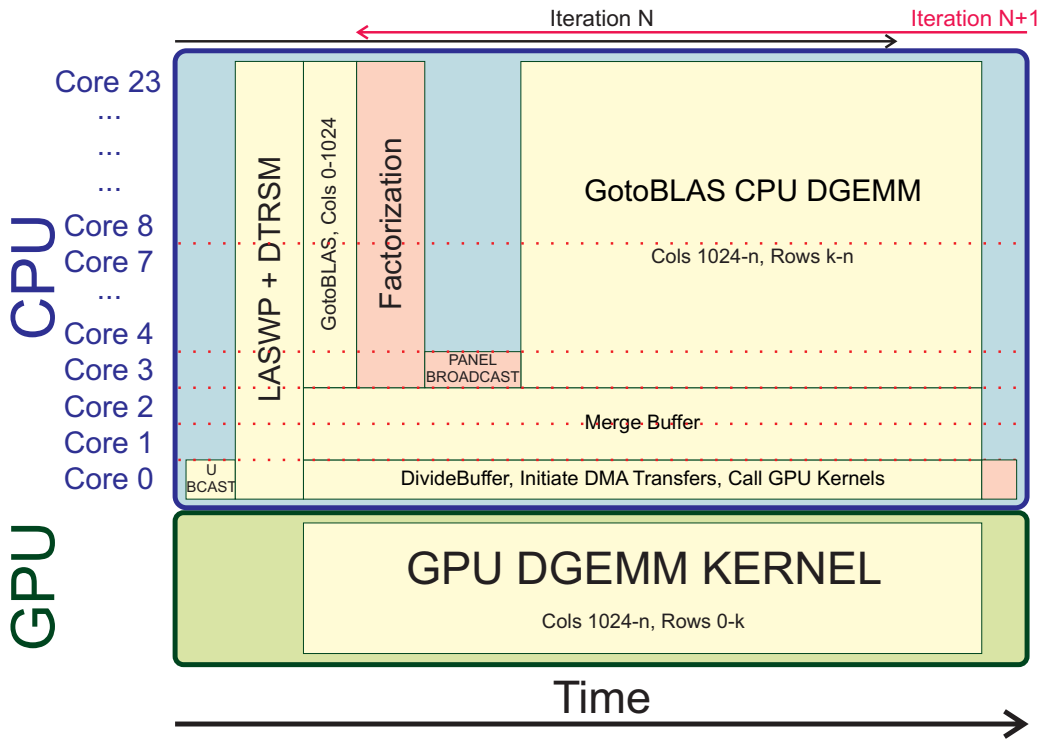

Figure 33: Process-Flow of GPU-based HPL with Lookahead 1 (i)

Unfortunately, this collides with the GPU/CPU ratio calculation in CALDGEMM, as the CPU is no longer only calculating DGEMM. The available time of CPU and GPU is different so the ratio can no longer be calculated considering only the performance. To account for this, in Linpack-Mode CALDGEMM continuously measures the CPU and GPU performance and the time required for the transfer and the factorization and adjusts its ratio using a filter incorporating the new data. With r the old ratio, g the GPU performance, c the CPU performance, t_g the GPU DGEMM time and t_c the CPU DGEMM time the new ratio r' is calculated as $r' = \frac{1}{2} \cdot (r + \frac{g}{c'+g})$ with $c' = \frac{ct_c}{t_g}$. As the factorization is not called each iteration in multi-node runs, but depends on the position inside the grid, CALDGEMM respects the grid position and maintains ratio parameters for usage with and without factorization. CALDGEMM continuously rechecks whether the ratio used was appropriate. If that is not the case it reinitializes the database. For the (re)initialization an empirically chosen penalty is applied to the CPU performance to correct for the factorization time.

Maximizing CPU Utilization In the first step the broadcast time was hidden, however, most CPU threads are still idling during the broadcast. This can be solved by starting an additional DGEMM with a thread count reduced by one. Here two possibilities exist:

- One CPU core is reserved exclusively for the broadcast. It is idling afterwards. The DGEMM can immediately start and process the entire matrix.
- Two DGEMMs are executed. A first one using all but one cores. This one core handles the communication. After the broadcast has finished a second DGEMM is started using all available cores.

Fig. 34 shows that the dynamic CPU scheduler that starts two DGEMMs is slightly faster.

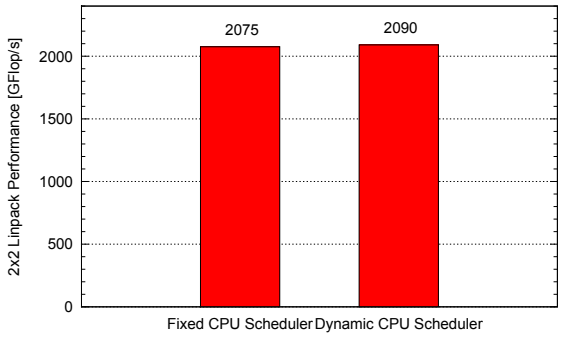

Figure 34: Performance using Dynamic CPU Scheduler

Some more details on the dynamic CPU scheduler shall be given. It is difficult to synchronize the first DGEMM to finish exactly when the broadcast completes. In fact it is impossible. What CALDGEMM does is: it continuously measures the broadcast time. It knows the CPU DGEMM performance and can thus guess how large the matrix for the first call should be, so that the call finishes shortly after the broadcast. Some extra time is added to correct for variations in the broadcast time. When CALDGEMM finds out that this first call would already process most of the matrix it does not split in two calls, because the second call would then process only a small matrix and thus be slower, although it could use an additional core.

In case the predicted broadcast time underestimated the real time the first DGEMM finishes before the broadcast. As it makes no sense to wait for the broadcast to finish, the second

DGEMM is then called without the additional core. Alternatively the second DGEMM could be started with the core while the broadcast is still running. But then one GotoBLAS thread would have a decreased performance what in the case of GotoBLAS can have a tremendous effect on overall performance.

Fig. 35 shows the improved lookahead with the dynamic CPU scheduler.

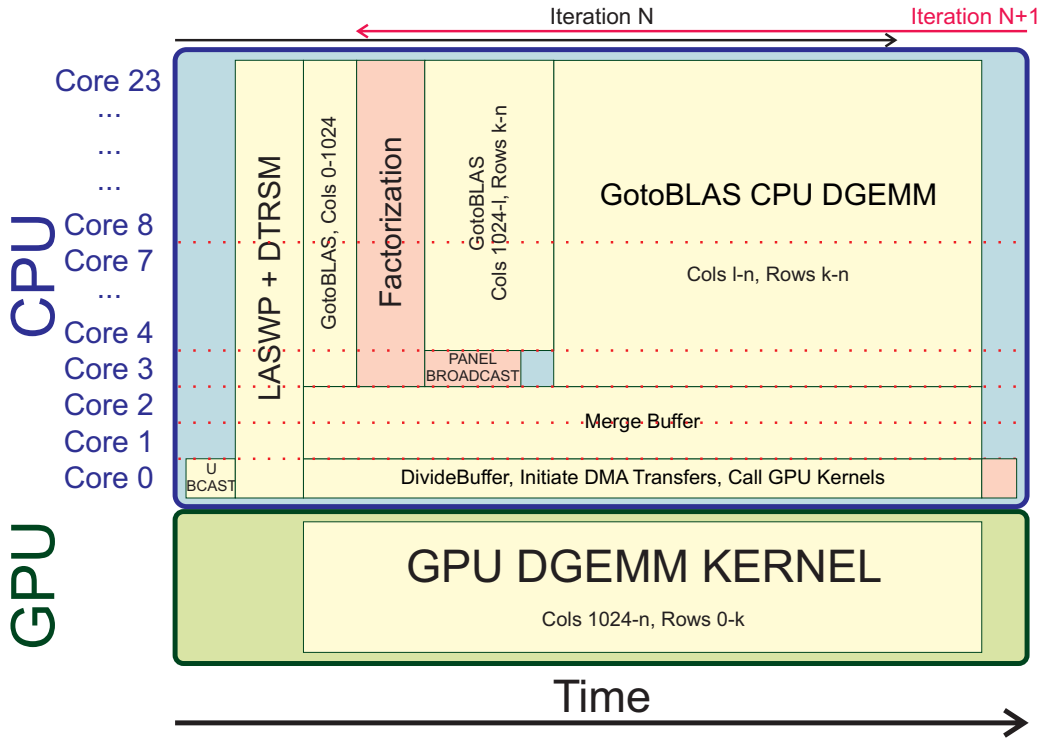

Figure 35: Process-Flow of GPU-based HPL with Lookahead 1 (ii)

Maximizing CALDGEMM GPU Performance Unfortunately, benchmarks revealed that the lookahead mode - as just explained - is inferior to the version without lookahead. It turned out that this is related to a decreased GPU DGEMM performance. The GPU contribution to the DGEMM drops from about 450 GFlop/s to only 400 GFlop/s. However, this is the effect on the overall GPU DGEMM performance. The factorization and the broadcast do not last for the entire DGEMM. But as long as they are active they reduce GPU DGEMM performance to about 300 GFlop/s. (When the broadcast has finished performance comes back to 450 GFlop/s resulting in an average of 400 GFlop/s.)

It turned out that the reason for this is a greatly reduced performance of the MergeBuffer routine due to memory congestion. This can be solved by using an additional merger thread, so increasing t from 2 to 3. This was implemented in a way, that the additional thread is only active during factorization and broadcast. Fig. 36 show a process-flow diagram.

Unfortunately, this approach did not solve the reduced overall performance at all. As expected the GPU DGEMM performance came back to normal. But the factorization and broadcast time increased by an order of magnitude and thus exceeded the DGEMM time. The problem just moved to another spot.

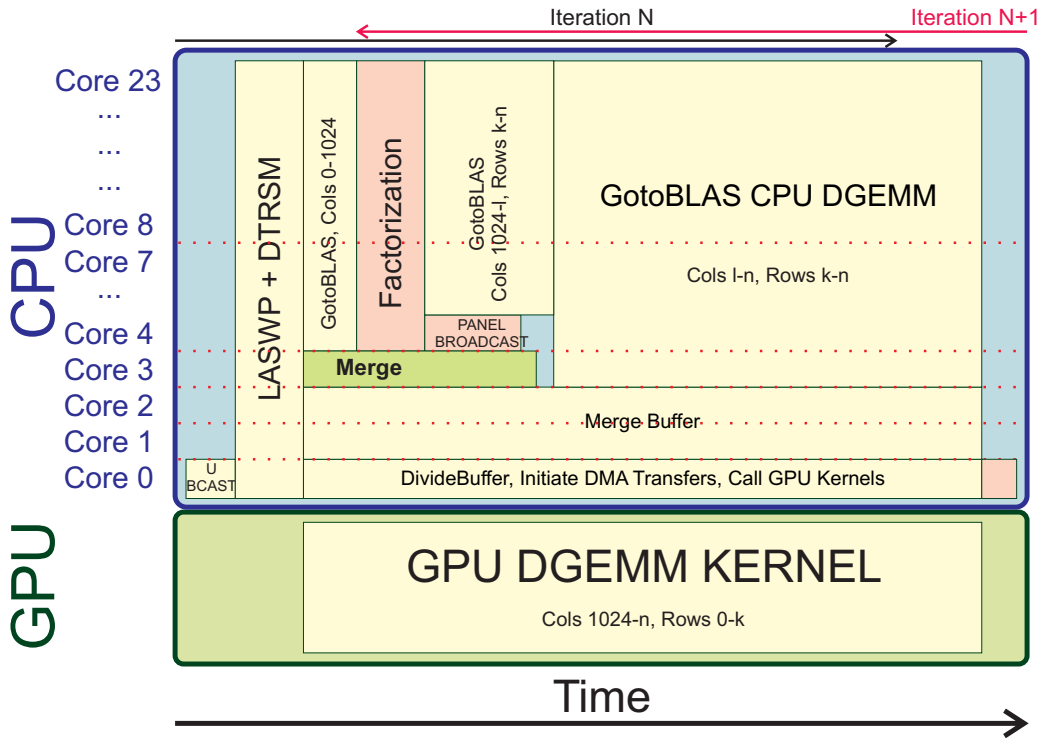

Figure 36: Process-Flow of GPU-based HPL with Lookahead 1 (iii)

Avoiding Memory Congestion using the AMD Driver Patch It shall be noted here, that the above tests were made before the driver hack was done. With the driver hack a new approach was followed. The driver hack allows for a reduction of the output thread count to a single thread while still achieving 620 GFlop/s DGEMM performance. Going back to the previous Lookahead version without the additional merge thread but using the driver hack helped, but did not solve the problem.

The full GPU performance was finally achieved by a reduction of the factorization thread count. Actually the factorization performance did no even suffer much. Before going to the process diagram again, the effect on GPU performance shall be analyzed. In the following the factorization thread count is generally reduced, so that only eight cores are active. This leaves $7 - t$ threads[28] for the factorization. As the driver hack is not generally available, also the performance without hack is analyzed. Fig. 37 shows the performance with and without lookahead throughout an HPL run. The x axis is the iteration number in HPL, so for high x the matrix size decreases. The x axis is not proportional to the time, as the first iterations need much more time than the later ones. Runs with the driver hack require only a single output thread. In contrast, versions without hack and two or three output threads respectively are shown.

It can be seen that with the driver hack and only a single thread both curves are very much alike. This means GPU DGEMM performance no longer suffers when enabling lookahead. For the curves without hack the GPU DGEMM performance is significantly lower when enabling lookahead. A confusing observation is, that for big x, the GPU DGEMM performance is even higher with lookahead than without. This is due to the fact, that only the raw GPU contribution to the DGEMM performance is shown here. With lookahead the CPU performs the factorization and the broadcast during the GPU DGEMM. At high x the factorization

[28] t is the number of concurrent output threads as defined in section 2.3.1.

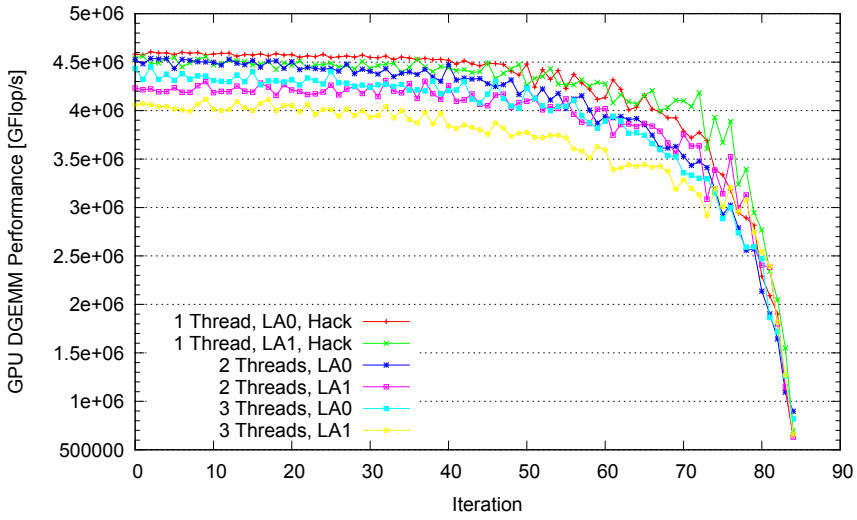

Figure 37: GPU Only DGEMM Performance for Lookahead during Linpack

takes up a significant amount of the iteration time. This reduces the part of the DGEMM processed by the CPU and increases the GPU part. However, a bigger GPU matrix can increase the GPU DGEMM performance as the pipeline can better hide the latencies. Finally, Fig. 38 shows that the combined GPU/CPU DGEMM performance is still higher without lookahead as it is expected.

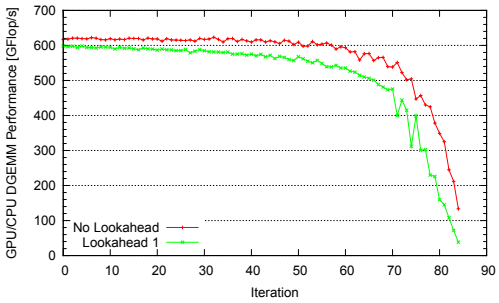

Figure 38: Total GPU/CPU DGEMM Performance for Lookahead during Linpack

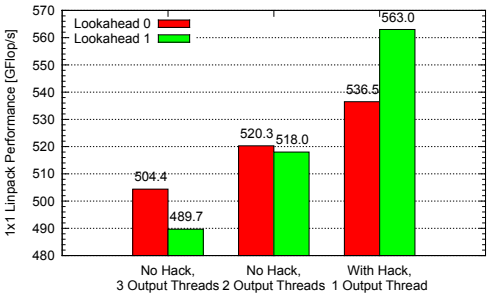

Figure 39: Lookahead Performance using AMD Hack

In combination with the driver hack and the reduced factorization thread count the lookahead is finally working well. Fig. 39 shows the performance with different output thread counts with lookahead and without. Finally Fig. 40 shows the process-flow with lookahead.

3.4.2 Lookahead 2

In the next step the time for the pivoting shall also be hidden. The DGEMM requires two processes to have finished: the swapping of the rows and the DTRSM which does the triangular solve on the U-matrix. However, it is possible to swap only the first x columns and run DTRSM only on the first x columns of the U matrix. Then the DGEMM can already process the first x columns of the C-matrix. This shows that the pivoting and the update can also be pipelined.

This is implemented in the way, that at first $nb + h_{\max} = 1024 + 4096 = 5120$ columns are processed. Afterwards even with $h = 4096$ the GPU can immediately start processing

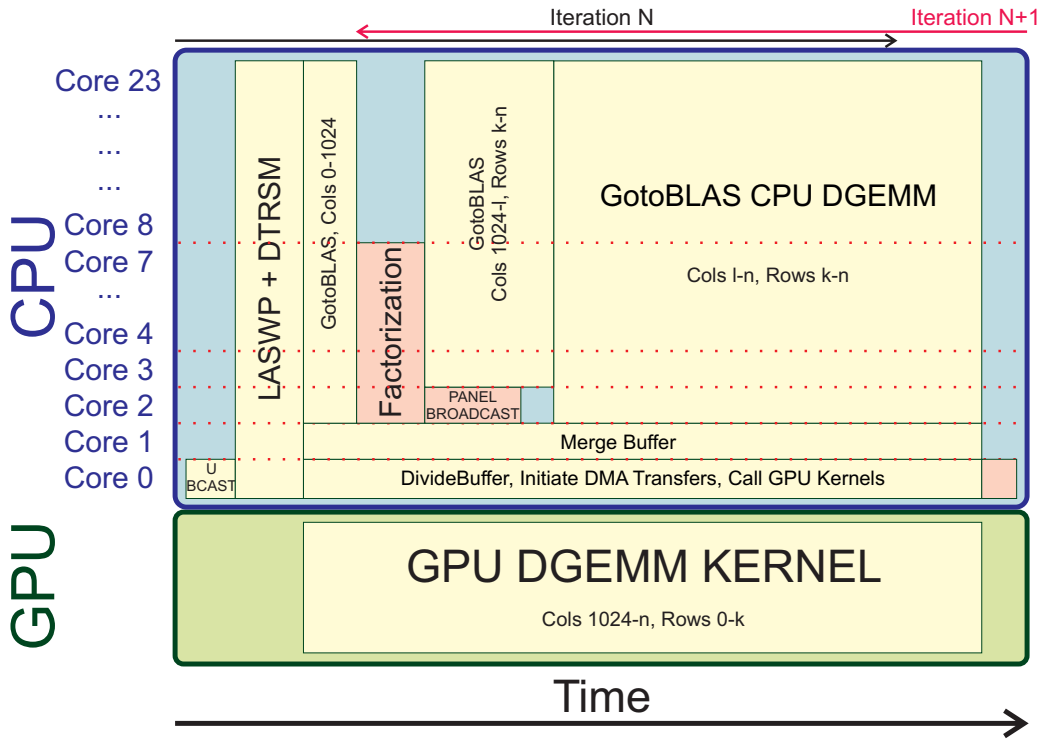

Figure 40: Process-Flow of GPU-based HPL with Lookahead 1 (iv)

columns 1024 to 5120. (The first 1024 columns are skipped as they are completely processed by the CPU for lookahead 1 afterwards.) Concurrently LASWP and DTRSM is iterated over the entire matrix. CALDGEMM checks, before it processes a tile, whether the swaps and DTRSM for the corresponding columns have already finished. The amount of columns processed in each LASWP/DTRSM iteration is continuously increased by a factor of two, as LASWP and DTRSM is faster for a bigger data.

The first lookahead 2 implementation suffered from the same problem as the initial lookahead 1 implementation. To avoid memory congestion the number of threads processing the LASWPs has also been reduced as for the factorization. As the LASWPs are memory bound it seems reasonable to employ all available memory controllers. This is done by using only even numbered cores, thus half of the cores at each die. (The version is called improved lookahead 2.) Fig. 41 Shows a process diagram with lookahead 2.

Fig. 42 shows the time required for each HPL iteration using lookahead 1 and lookahead 2. It can be seen that they are very much alike. Fig. 43 shows the time difference. It reveals that lookahead 2 is faster at the beginning of the linpack run, but gets slower towards the end. Therefore, a mixed lookahead 2 was implemented, that switches back to lookahead 1 where the new variant gets slower.

Fig. 44 shows the performances for the multiple lookahead 2 implementations. Only the mixed implementation can slightly outperform the lookahead 1 code. Finally, Fig. 45 shows a comparison of single-node and multi-node runs without lookahead and with lookahead 1 & 2. It can be seen that also the single-node performance improved. This is due to the fact that not only broadcast time but also factorization and LASWP time is hidden. In the end the lookahead 2 did not bring the significant improvement that was expected. Reasons for this will be analyzed in the next section.

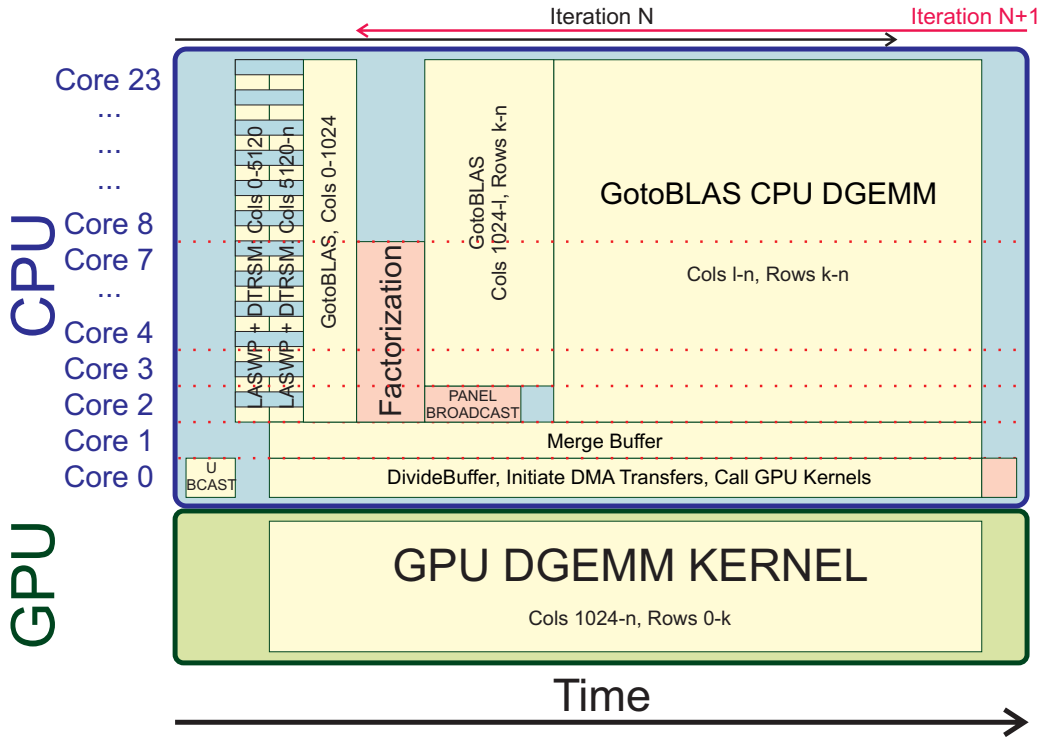

Figure 41: Process-Flow of GPU-based HPL with Lookahead 2

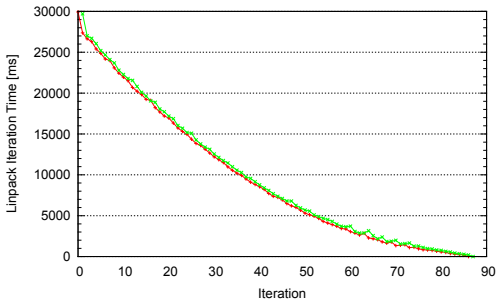

Figure 42: Iteration Times during Linpack with Lookahead 1 / 2

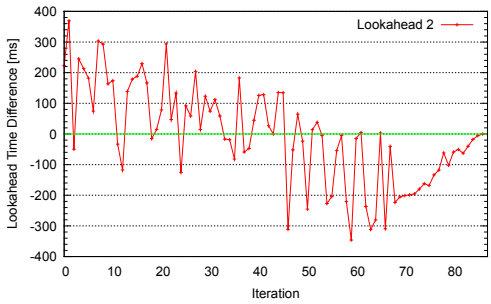

Figure 43: Iteration Time Difference

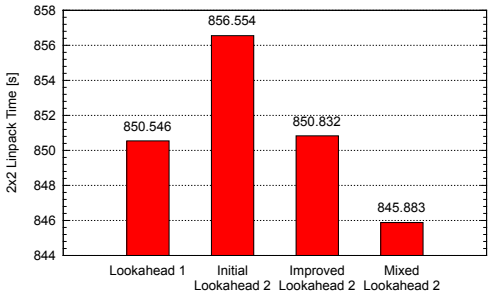

Figure 44: Performance of different Lookahead 2 Implementations

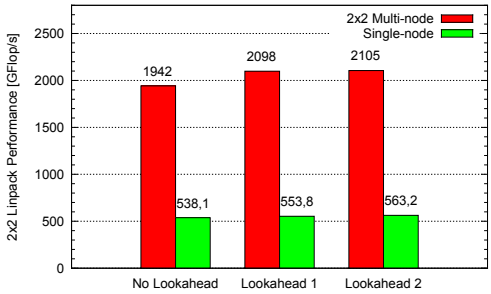

Figure 45: Performance of different Lookahead Modes

3.4.3 Performance Analysis

Finally, in this section it shall be analyzed how well the CALDGEMM features work within HPL. Also the effects of the lookahead optimizations will be discussed further.

CALDGEMM Performance in HPL At first an overview over the CALDGEMM scheduling efficiency shall be given. The splitting location for the first phase CALDGEMM GotoBLAS run cannot be chosen arbitrarily. The exact location lies inside a row of tiles. Therefore, it is very probable, that when the first phase is finished there are still tiles unprocessed. Applying the same ratio again for the second should assign tiles within this row of tiles to the CPU. If the ratio is chosen optimally, obviously no third phase run is needed. Fig. 46 visualizes the scheduling in an HPL run without lookahead. It shows the number of rows of tiles processed in the second phase and the total number of tiles processed in the third phase. The scheduling is considered optimal, when the second phase value is one and the third phase value is zero. However, as the exact splitting position jumps randomly, even for an optimal scheduling the phase two value should drop to zero sometimes. Applying these criteria the figure reveals that the scheduling is mostly optimal.

In contrast, Fig. 47 shows the same plot using lookahead 2. It can be seen that the second phase value is still quite optimal. In contrast, the third phase value has a broader distribution. The reason is that the parallel factorization and swapping introduce additional inherent inaccuracy into the ratio calculation. Anyway, still the third phase value drops to zero for quite some iterations. Therefore, the scheduling can still be considered optimal. To lower the amount of third phase CPU tiles the ratio must be adjusted for a faster CPU. But then, in the iterations without third phase CPU runs, the GPU would probably have to wait for the CPU wasting quite some time.

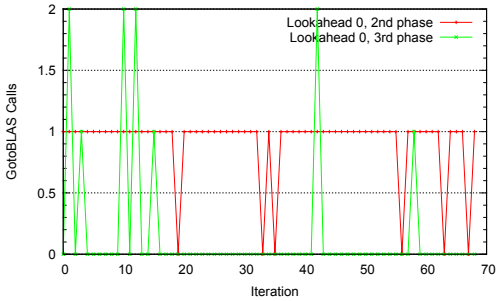

Figure 46: Analysis of CALDGEMM Scheduling Efficiency without Lookahead

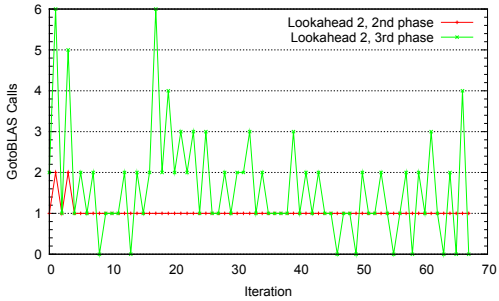

Figure 47: Analysis of CALDGEMM Scheduling Efficiency with Lookahead 2

For a further scheduling analysis Fig. 48 shows the total GPU utilization time, the total CPU utilization time and the CPU DGEMM time. (Obviously the difference between total CPU time and CPU DGEMM time is the sum of factorizations, LASWP and DTRSM time.) One can see a jump in the CPU DGEMM time, this is exactly at the position where the mixed lookahead switches to mode 1. Afterwards the LASWP and DTRSM time is excluded from the plot, as it lies outside the CALDGEMM call.

Finally, the total CPU and total GPU time are almost the same. Fig. 49 shows the difference of the two. For processing one tile (assuming $h = 4096$) the GPU requires below 75 ms, the CPU below 230 ms. Thus, the scheduling can be improved as soon as the difference is above 230 or below -75. It can be seen that the latter is never the case. In total, the difference reaches up to 250, which, however, does not allow for much optimization. It should also be

noted, that at many points the GPU performance is overestimated, what is reflected here: The CPU idle time is higher than the GPU idle time.

In addition to the time difference, the GPU wait time is shown. This is the amount of time the GPU thread is really waiting for the CPU thread's mutex to unlock. Optimally they should be of equal absolute value. This is, however, not completely ensured because the synchronization is not perfect. Still, the numbers agree with only small deviations.

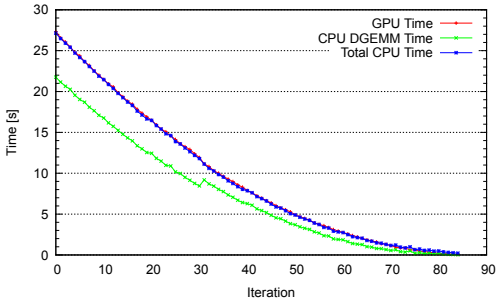

Figure 48: Analysis of CALDGEMM Ratio Calculation with Lookahead 2

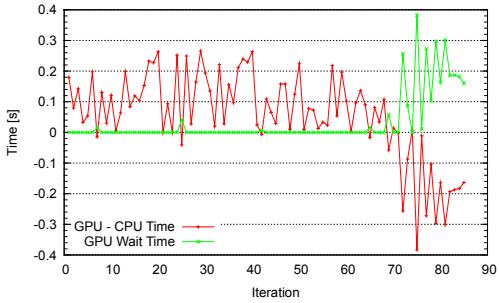

Figure 49: Difference in GPU/CPU Time during Linpack with Lookahead 2

HPL Overall Performance Recall that the lookahead 2 performance was a bit disappointing. A reason is given by Fig. 50. The LASWP and DTRSM time increases significantly when executed in parallel to the DGEMM.
Fig. 51 shows the same data for the factorization time. In comparison with lookahead 2, lookahead 1 also hides the broadcast not only the factorization thus resulting in greater speedup.

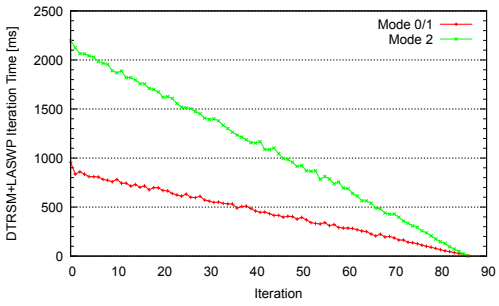

Figure 50: DTRSM + LASWP Time during Linpack

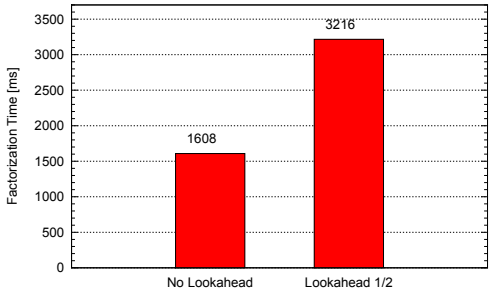

Figure 51: Factorization Time during Linpack

Finally Fig. 52 shows how the multi-node Linpack performance evolved over the time with more and more patches.

4 Torture Tests

Measuring the GPU temperature revealed, that the kernel that comes close to peak performance can easily overheat GPUs. For all the tests, both system and GPU fans were pinned to 100%. However, this opens the possibility to use CALDGEMM as a torture test.

CALDGEMM should be superior to single-node HPL in this discipline, because it can keep executing GPU kernels all the time, whereas HPL is pausing GPU execution from time to time.

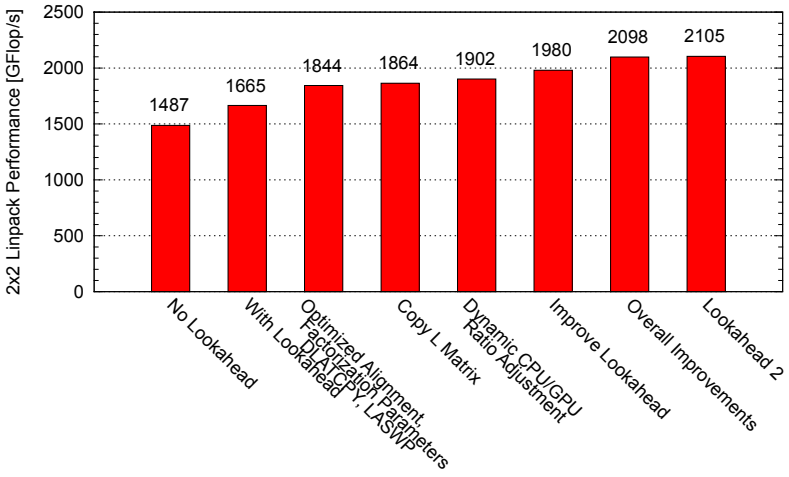

Figure 52: Linpack Performance Evolution Summary

The CALDGEMM benchmark / torture test was originally supplied with two initialization methods, a fast method that initialized all matrices with zero; used only for performance tests. And an equally distributed linear congruence random number generator (RNG) in the range $(0 : 1)$. Fig. 53 unveils an interesting fact: The single-node HPL causes much more heat on the GPU than CALDGEMM using the RNG. CALDGEMM using the zero initialization is way behind. This is due to the fact that no bits are flipped in the GPU.

The reason for CALDGEMM to stay cooler than single node HPL turned out to be the random number generator. The CALDGEMM RNG range was changed to $(-0.5 : 0.5)$ and later to $(-10 : 10)$. Both brought an improvement but HPL still got hotter. As HPL multiplies matrices that are already altered in the factorization process, the HPL matrices are not in the range $(-0.5 : 0.5)$ even though the HPL RNG produces numbers in this range. The factorization makes the entries in the HPL matrix rather gaussian distributed around zero.

A RNG with a gaussian distribution with estimation value zero and variance 25 was implemented into CALDGEMM, which finally resulted in the same temperature than single-node HPL. No possibility was found to optimize the torture test any further.

Fig. 54 shows the torture test with gaussian RNG and single node HPL on different nodes. It can be seen that the temperature of both benchmarks is the same, however, the temperatures between the nodes differ tremendously, with some nodes overheating very soon, while others stay below 85°C.

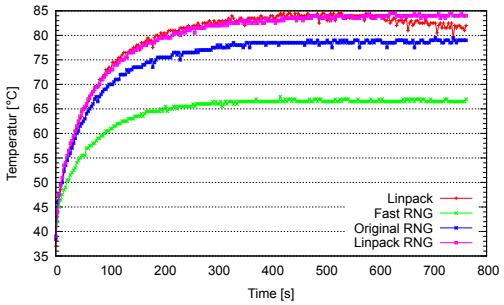

Figure 53: Heat produced by Linpack and different Torture Tests

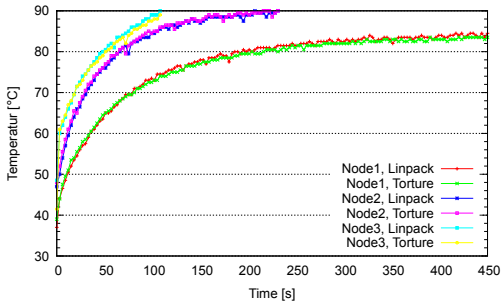

Figure 54: Temperature of Linpack and Torture Test for different Nodes

5 DGEMM & Linpack Performance

In summary, Fig. 55 shows the peak performance per node reached in various disciplines. Tab. 56 shows the numbers again with the efficiencies related to the theoretical peak performance of the device. The kernel itself can utilize above 90% of the GPU peak performance. The DGEMM can utilize above 80% of the combined GPU/CPU peak performance. Finally, the single-node HPL can still achieve more than 75% of the theoretical peak performance. When it comes to network efficiency (Tab. 57) shows that the lookahead is able to almost conserve the single node performance. The multi-node version looses less than 7%. Finally Tab. 58 shows that the HPL scales well to many-node systems. The efficiency does not suffer significantly when going from 4 to many-node. The number reported to the Top500 list is **285.2 TFlop/s** which made the LOEWE-CSC ranked place **22** in the Top500 and place **8** in the Green500 in November 2010.

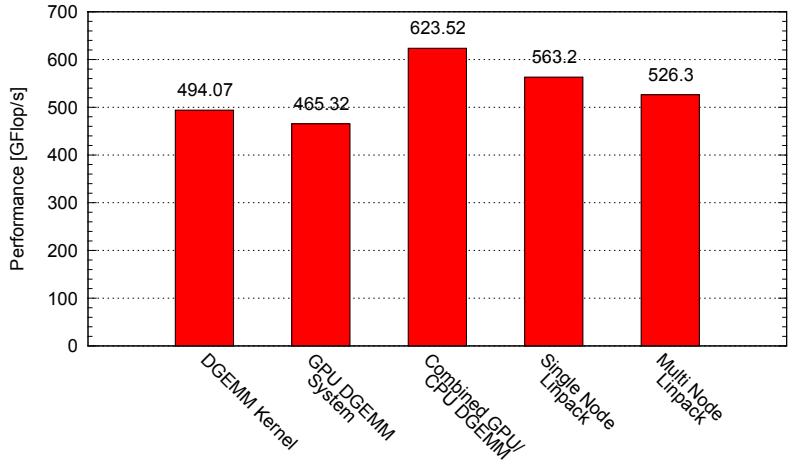

Figure 55: Peak Performances achieved with CALDGEMM / Linpack

Discipline	Performance Reached	Peak Performance	Efficiency
DGEMM Kernel	494.07	544.00	90.93%
GPU DGEMM	465.32	544.00	85.54%
GPU/CPU DGEMM	623.52	745.60	83.63%
Single-node HPL	563.20	745.60	75.54%
Multi-node HPL (2x2)	526.25	745.60	70.58%

Table 56: Peak Performance and Efficiency per Node

Discipline	Multi-Node Performance	Peak Performance[29]	Efficiency
Single-node HPL	563.2	563.2	100.00%
Multi-node HPL (2x2)	2105.0	2252.8	93.43%

Table 57: Network Efficiency

[29]The peak performance for the network efficiency is not calculated by the accumulated peak performances of the nodes, but by the highest linpack performance achieved in single-node runs times the number of nodes.

Discipline	Performance Reached	Peak Performance	Efficiency
Single-node HPL	563.2	745.6	75.54%
Multi-node HPL (2x2)	2105.0	2982.4	70.58%
Many-node HPL	**285200**	409248	69.69%

Table 58: Multi-Node Performance

6 Perspective: Multinode Linpack, Lookahead 3

Considering the improvements with lookahead 2 it is questionable whether performance can be increased significantly by hiding the U-broadcast time. Still, this is an option for the future. In addition, the free CPU cores during factorization and LASWP could be used to contribute to the large CPU DGEMM. This should be possible, as it was already shown that a CPU DGEMM does not slow down the GPU. It is unclear, however, what will happen when running factorization and DGEMM in parallel.

A problem arising here is that the GotoBLAS library is currently not capable of running independent multiple tasks simultaneously. The GotoBLAS patch would have to be improved. To be able to hide the U-broadcast some more parts of the DGEMM would have to finish earlier, e.g. the first 1024 columns.

A different subject that can definitely increase the performance is the usage of multiple GPUs. Then a second core should be used for a second MergeBuffer process.

Fig. 59 shows a process-flow with some ideas for the future.

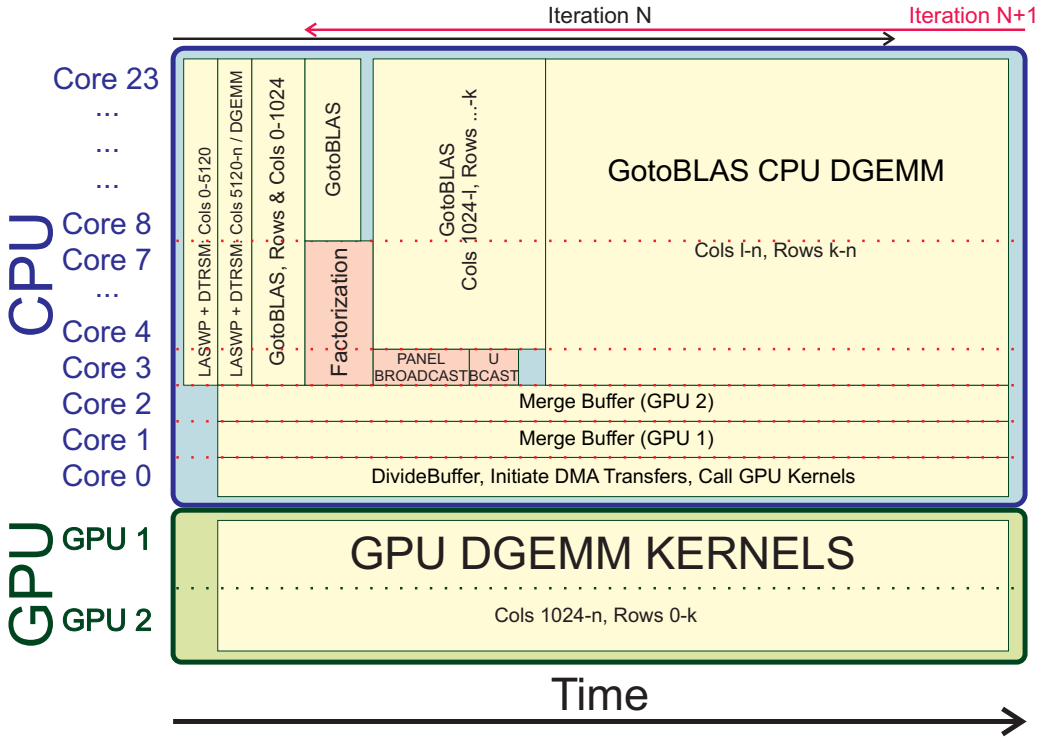

Figure 59: Process-Flow of multi-GPU HPL with Lookahead 3

Bibliography

[Amd1] Advanced Micro Devices, "AMD Stream Computing Guide",
http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf.

[Amd2] Advanced Micro Devices, "ATI Compute Abstraction Layer Intermediate Language Reference Manual",
http://developer.amd.com/gpu_assets/ATI_Intermediate_Language_%28IL%29_Specification_v2c.pdf.

[Amd3] Advanced Micro Devices, "ATI Evergreen Family Instruction Set Architecture Reference Manual",
http://developer.amd.com/gpu/ATIStreamSDK/assets/AMD_Evergreen-Family_ISA_Instructions_and_Microcode.pdf.

[Amd4] Advanced Micro Devices, "AMD Core Math Library",
http://developer.amd.com/cpu/Libraries/acml/Pages/default.aspx.

[Cop1+] D. Coppersmith, S. Winograd, "Matrix multiplication via arithmetic progressions", 1990, Journal of Symbolic Computation 9 (3): 251–280,
http://www.cs.umd.edu/~gasarch/ramsey/matrixmult.pdf.

[Hpl] University of Tennesse, "High Performance Linpack Algorithm",
http://www.netlib.org/benchmark/hpl/algorithm.html.

[Int2] Intel Corporation, "Intel Threading Building Blocks Reference Manual , July 2009",
http://www.threadingbuildingblocks.org/uploads/81/91/LatestOpenSourceDocumentation/Reference.pdf.

[Int4] Intel Corporation, "Intel Press Release 31.5.2010", http://www.intel.com/pressroom/archive/releases/20100531comp.htm.

[Loe] Goethe University of Frankfurt, Center for Scientific Computing, "LOEWE-CSC Cluster",
https://csc.uni-frankfurt.de/csc/index.php?id=51.

[Nak] N. Nakasato, "A Fast GEMM Implementation On a Cypress GPU", University of Aizu, 2010

[Roh1] D. Rohr, "ALICE TPC Online Tracking on GPGPU based on Kalman FIlter", Diploma Thesis, University of Heidelberg, 2010

[Str] V. Strassen, "Gaussian Elimination is not Optimal", Numerische Mathematik, 13 (1969) 354-356.

[Tac] Texas Advanced Computing Center, GotoBLAS Basic Linear Algebra Library, `http://www.tacc.utexas.edu/tacc-projects/`.

[Top1] Top 500 Supercomputing Sites, `http://www.top500.org/`.

[Vol+] V. Volkov, J. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra", SC '08 ACM/IEEE conference on Supercomputing Proceedings (Piscataway, NJ, USA, 2008), pp. 1–11.