Analysis of a memory bandwidth limited scenario for NUMA and GPU systems

Jens Breitbart
Research Group Programming Languages / Methodologies
Universität Kassel, Germany
Email:jbreitbart@uni-kassel.de

Abstract—The processing power and parallelism in hardware is expected to increase rapidly over the next years, whereas memory bandwidth per flop and the amount of main memory per flop will be falling behind. These trends will result in both more algorithms to become limited by memory bandwidth, and overall memory requirements to become an important factor for algorithm design. In this paper we study the Gauß-Seidel stencil as an example of a memory bandwidth limited algorithm. We consider GPUs and NUMA systems, which are both designed to provide high memory bandwidth at the cost of making algorithm design more complex. The mapping of the non-linear memory access pattern of the Gauß-Seidel stencil to the different hardware is important to achieve high performance. We show that there is a trade-off between overall performance and memory requirements when optimizing for optimal memory access pattern. Vectorizing on the NUMA system and optimizing to utilize all processors on the GPU does not pay off in terms of performance per memory used, which we consider an important measurement regarding the trends named before.

I. Introduction

The last years saw a great increase in processing power being made available by multicore CPUs and accelerators. However, in contrast to processing power, memory bandwidth and the amount of main memory is increasing rather slowly. Future predictions show these trends increasing. For the first Exa-flop system being expected sometime around 2018, a decrease by about a factor of 100 compared to current systems is expected for both node memory bandwidth per flop and the amount of node memory per flop. Furthermore, the level of concurrency in one node is expected to increase by a factor of 100 - 1000 [1]. These trends may lead to yet hardly discussed trade-offs between memory and computation usage. With the decrease of memory per flop, it may e.g. no longer be possible to sustain all of the processing elements with data, since there is simply not enough memory available. The same may happen with regard to the available memory bandwidth, so that only a fraction of the available processing elements will be active. Two prominent concepts providing for high memory bandwidth are NUMA systems using multiple CPUs to multiply the available memory bandwidth per node, and GPUs with memory controllers designed for specific memory access patterns. Both concepts, however, suffer from different problems. On NUMA systems, the cost for memory accesses depends on the access pattern that is being issued by a set of threads.

In this paper we analyze both architectures with the Gauß-Seidel stencil, which has the following properties:

- performance is limited by memory bandwidth
- the memory access pattern is non-linear
- parallelization is non-trivial and does not allow for a linear speedup.

These characteristics make the algorithm a good indicator for the future usability of both architectures.

We studied the algorithm on the following hardware:

- single socket Nehalem CPU (Core i7-920, 4-cores) referred to as SMP system
- four socket Nehalem CPU (4 x Xeon X7560, 8-cores each) referred to as NUMA system
- NVIDIA GeForce GTX 280 (GT200 chip)
- NVIDIA Tesla C2050 (GF100 chip)

Nehalem CPUs are Intels latest x86 CPU. The SMP system has 4 cores with 3 memory channels. The NUMA system uses four 8-core CPUs, each supplied with 4 memory channels. How data is distributed among the local memories can be configured by the developer.

NVIDIA GPUs are tile-based many-core systems with their own memory subsystem highly optimized for throughput. The GT200 puts rather strict constraints on the memory access pattern to achieve high bandwidth (so-called coalesced memory accesses), whereas the GF100 provides caches to reduce the constraints. Both GPUs offer fast on-chip memory.

The Gauß-Seidel stencil works on a two-dimensional matrix, repeatedly running through the matrix and updating the current value based on its four neighbors. The GT200 puts rather strict constraints on the memory access pattern to achieve high bandwidth (so-called coalesced memory accesses), whereas the GF100 provides caches to reduce the constraints. Both GPUs offer fast on-chip memory.

The Gauß-Seidel stencil works on a two-dimensional matrix, repeatedly running through the matrix and updating the current value based on its four neighbors. The GT200 puts rather strict constraints on the memory access pattern to achieve high bandwidth (so-called coalesced memory accesses), whereas the GF100 provides caches to reduce the constraints. Both GPUs offer fast on-chip memory.

Our implementations do not strictly follow the waveform pattern, but a thread may go ahead if all required data is available. Atomically incremented counters are used for pair-wise synchronization. For the CPU systems, we studied two ways of mapping the data matrix to threads, which result in two different memory access patterns. Splitting by rows results in memory reads of one thread to follow cache
lines, whereas splitting by columns does not. Furthermore we studied two memory layouts: row-major layout, and a diagonal-based scheme in which diagonals of the matrix form a linear memory block. The diagonal-based scheme requires more memory due to padding for a reasonably simple addressing scheme. For GPUs, the different mapping has no influence on performance, so we only discuss the different memory layouts. It is required to be able to use the SIMD units of the CPU. Table I shows all versions in detail.

On all test systems, a diagonal based implementation provides the best overall performance. However, this version falls behind when measuring performance per used memory. With main memory per flop decreasing, we consider performance per used memory an important factor to allow scalability for future hardware and large problem sizes.

The SMP system hardly reacts to the different data to thread mappings, whereas on the NUMA system the mapping must be coordinated with the memory distribution to achieve a reasonable speedup.

The diagonal based memory layout version is about 20% faster than the row-major version on the GF100, as it allows for high GPU processor (warps) utilization. However in terms of performance per memory used the row-major version outperforms the diagonal one, despite utilizing only a fraction of the available processors. On the GT200 both versions perform about the same in terms of performance per memory used, which is a result of GT200 slow performance with the row-major layout. There is hardly an increase in best performance comparing the GF100 and GT200, which may indicate that memory bandwidth on GPUs is not increasing as rapid as before.

Overall, the NUMA system provides the best performance of all systems with regard to performance and performance per memory used. The GPUs provide 50% more performance than the single socket CPU system, however only provides a relatively small amount of memory.

The paper is organized as follows. First, Sect. II gives a more detailed overview of the used hardware architectures, followed by an introduction to the Gauß-Seidel stencil in Sect. III. The next two sections detail the algorithm and the implementation at the CPU (Sect. IV) and the GPU (Sect. V) systems. Section VI discusses the performance difference of the CPU and GPU based systems. The paper finishes with an overview of related work and conclusions, in Sects. VII and VIII, respectively.

II. HARDWARE SETUP AND PROGRAMMING SYSTEM

We used two fundamentally different hardware architectures: CPUs and GPUs. CPUs have traditionally been optimized for sequential computation allowing rather efficient random memory accesses, whereas GPUs are massively parallel, throughput-oriented devices putting strict constraints on memory access patterns.

The CPUs in both the SMP and NUMA systems are based on Intel’s Nehalem architecture. Nehalem is Intel’s first x86 architecture that features an on-chip memory controller to decrease memory access latency and increase memory bandwidth. All CPUs share a common SIMD instruction set working on 128 bits of data, so e.g. two 4-float vectors can be added together in a single instruction. To achieve high performance when using the SIMD units, one must use special instructions to load 128 bits of consecutive memory into a register. The system provides two instructions for that: MOVAPS requires a memory alignment of 16 for the data to be loaded, and outperforms MOVUPS, which is able to load from any memory alignment.

The developer can configure how memory is distributed among the local memories on the NUMA system. By default, it is split into stripes of size 256 that are distributed in a round-robin fashion (striped memory distribution). This concept equally saturates the system memories with data, however optimizing for the stripe size is most likely not portable. Furthermore, a single core can only work on a relatively small amount of linear memory without the need to access remote memory. Alternatively, the system can be configured so that memory is local to the allocating thread (local memory distribution), and developers have direct knowledge of where data is stored.

Table II shows the performance of the STREAM memory benchmark on the CPU systems run with 4 threads (SMP) and 32 threads with stripe memory layout (NUMA). We used the following hardware for our experiments: The SMP system uses a Core i7 920 with 3 memory channels supplied with PC1066 memory. The Core i7 920 has 4 cores, each with hardware support to run 2 threads effectively (SMT). The NUMA system uses four Xeon X7560, each with 4 memory channels supplied with PC1066 memory. A single socket has 8 cores and SMT has been disabled. The four socket NUMA system thereby has a total of 32 cores with 16 memory channels.

GPUs are tile-based many-core systems on which a set of processors are bundled together in a tile called streaming multiprocessor (SM). The processors in one tile are SIMD units, so all processors of a tile must execute the same instruction at a time. On the older GT200 240 cores are organized into 30 SM (8 cores per SM), whereas the new GF100 has 448 cores organized into 14 tiles (32 cores per SM). A SM provides fast on-chip memory on which all processors of the SM can work together. The GF100 has a L2 cache shared by all SMs, whereas each SM has its own L1 cache. The size of the L1 cache can be configured by the user. It can be chosen from 48 KB of L1 cache, which results in 16 KB of shared memory, or 16 KB of L1 cache, which gives 48 KB of shared memory. The GT200 only provides 16 KB of shared memory.

NVIDIA GPUs are programmed in CUDA. Developers start so-called kernels, which are similar to main functions
of CPU programs. A kernel is executed by many threads, bundled in threadblocks with a maximum size of 512 (GT200) or 1024 (GF100) threads. Multiple threadblocks are scheduled onto a SM. Threads in one threadblock can utilize shared memory and can be synchronized with a fast barrier operation. The number of threadblocks that can be scheduled onto the same SM depends on the amount of resources a threadblock requires, e.g. if a threadblock requires 8 KB of shared memory, only two blocks can be scheduled onto the same GT200 SM. Developers strive to maximize the occupancy of the SMs, as the hardware uses parallelism to hide memory access latency.

The SIMD nature of the hardware is exposed to the developer as the so-called single instruction multiple threads (SIMT) model. In SIMT, a group of threads (called warp) is executed at a SM in SIMD-lockstep. The hardware handles any diverges in control-flow within a warp and guarantees for correctness, however this results in slower performance. On both GPUs a warp consists of 32 consecutive threads of one threadblock. The efficient memory access pattern required by the hardware are defined on a per-warp basis. To effectively utilize the bandwidth on GT200, threads of a warp must access data within a range of 128 bytes. If all but one thread access the same 128 byte range, the hardware will load 256 byte from off-chip memory and throw away all data that is not used, which effectively reduces the bandwidth. The GF100 off-chip memory accesses are cached, so for example in the same scenario as before the first warp would force two 128 byte cache lines to be fetched from off-chip memory, however if a second warp would access one of the two cache lines the data would not be fetched from off-chip memory, but read from cache. By concept, the hardware hardly supplies one cache line per thread, so data locality must be exposed between threads.

### III. GAUSS-SEIDEL STENCIL

The Gauß-Seidel stencil works on a two-dimensional data matrix $V$ with border elements fixed. The non-border element $(x, y)$ is in the $k$th iteration calculated by

$$V_{x,y}^k = \frac{V_{x-1,y}^k + V_{x+1,y}^k + V_{x,y+1}^{k-1} + V_{x,y-1}^{k-1}}{4}.$$  

This calculation is repeated for a fixed number of steps or until convergence. The algorithm has a low arithmetic intensity, as every non-border element is accessed five times and only 4 arithmetic operations are required to compute a new value of an element. Therefore, it should come as no surprise that the application-time is limited by memory bandwidth and not by the available processing power. Previous work [3] of the same authors discusses the performance characteristics of the Gauß-Seidel stencil in detail.

Figure 1 shows a visualization of the data dependencies of the Gauß-Seidel stencil. It is important to notice that the upper and the left values are from the current iteration, whereas the right and bottom values are from the previous iterations. The parallelization requires using the wavefront...
pattern [4], see Fig. 2. In a wavefront, the data matrix is divided into multiple diagonals. The elements of the same diagonal can be calculated in parallel.

The implementations follow a similar scheme for all architectures: We do not strictly follow the wavefront pattern, but use one counter per column/row to indicate how much of it has already been updated in the current iteration. Moreover, we split the matrix into multiple rows or columns and assign these in a round-robin fashion to threads. The counters are shared among all threads and busy waiting is used in case one thread must wait. We use a barrier after every iteration over the whole matrix.

There are a total of four variants of our base algorithm, two of each using the same memory layout (see Tab. I). One memory layout is the traditional row-major matrix layout (Fig. 2), the other one is a diagonal based one (Fig. 3). The diagonal based scheme results in linear memory reads for both GPUs and CPU SIMD units. However, the diagonal scheme also requires twice the amount of memory to allow for a reasonably simple addressing scheme. We implemented both mappings rows and columns of \( V \) to threads for CPU systems, so e.g. there is a version using diagonal based memory layout, which maps rows to threads.

The performance of the Gauß-Seidel stencil is measured in stencils per second, where a stencil corresponds to one update of \( V_{x,y} \).

IV. CPU IMPLEMENTATION

In this section, we first detail the SMP implementation, and discuss the differences to the NUMA implementation afterwards. The section closes with a performance comparison.

CPUs use multiple levels of fast on-chip cache to overcome the penalty of main memory latency and bandwidth. Caches, however, are only useful if the program exposes data locality, so reoccurring accesses to the data may be fetched from cache instead of main memory. A common way to increase cache utilization is tiling. It maximizes cache usage by subdividing loop iteration space into blocks. Since the Gauß-Seidel stencil is limited by memory bandwidth, tiling greatly increases its performance.

For the Gauß-Seidel stencil, tiling is applied in both the x- and y-dimensions, so we create two-dimensional tiles. The parallelization of the stencil is based on the tiles. We implemented both a mapping of rows and a columns to threads. We describe only the column based one, as the row based one is identical, except that rows are assigned to threads. We assign a column of tiles to each thread in a round-robin fashion, and use atomically updated counters as has been described before. Thus, a counter is updated whenever a tile has been calculated. Before the next tile, the counter of the preceding column is checked, and possibly rechecked in busy waiting. Therefore, smaller tiles increase the number of atomic operations, and larger tiles improve cache utilization.

Caches, in contrast to e.g. manually managed on-chip memory, are relatively easy to use, as one must only rearrange the memory access pattern as opposed to explicitly managed data movement. However, they also result in behavior not totally obvious to the user at first sight. For example, a naive implementation may suffer from false sharing in the counter variables, i.e. different cores modify different values in the same cache line. We use padding to solve this issue and store a single counter in one cache line.

The way caches are organized has also an impact on if one should assign rows or columns to threads. If rows are assigned to threads, the reads of the threads follow cache lines, whereas if columns are assigned to threads the reads do not.

To utilize the SIMD units of the CPU, we modified the algorithm to compute four elements at once. Figure 4 shows which elements must be loaded from memory and stored in vectors. To calculate the Gauß-Seidel stencil we must compute \( c = (a + b + d + e)/4 \). Note that this algorithm requires using the diagonal layout, as otherwise we could not load/store 4 elements at once and using individual loads/stores to/from SIMD registers would not allow for any speedup. This algorithm can compute 4 elements with the same number of instructions previously required to compute a single element. Figure 4 shows that three vectors (b, c, d) have the same memory alignment, whereas two differ. We made sure that these three vectors have a memory alignment of 16, which allows us to use the high performance vector load/write operations for these three vectors and the slower one for the other two. Note that all diagonal based CPU implementations use this SIMD algorithm.
The NUMA implementation requires to consider how memory is distributed among the sockets additional to all SMP related issues. The memory distribution must be coordinated with the memory access pattern to make sure that hardly any remote memory is accessed. By concept, the column based access pattern fits well to the stripe based memory distribution, whereas the row based distribution fits to the local memory distribution. Achieving optimal performance on the stripe based distribution requires adjusting the tile size to the size of the NUMA stripes including correct thread to socket assigning. As the NUMA stripes are rather small it hardly makes any sense to divide them among multiple cores, so one should use a round robin scheduling of threads to sockets so each threads works on a local stripe. This rather complex scheduling is strongly bound to the memory access pattern and the memory distribution.

When using the local memory distribution the location of the memory is completely transparent to the user, as one must only make sure that the thread allocating the memory will use the memory as well. This by concept may require parallel initialization of the program.

The SIMD algorithm by itself is not influenced by the memory distribution. Striped memory distribution allows to only work on small amounts of consecutive memory, and regularly requires us to load one of the four vector elements from remote memory, which is not very efficient. The local memory distribution on the other hand allows to not only read single elements from remote memory, but whole vectors, which is expected to be more efficient.

Figure 5 shows the performance on the SMP system of the diagonal and row-major versions with both memory access patterns. We can see that the diagonal version hardly scales from 2 to 4 threads. This is a clear indication that the implementation already consumes almost all of the memory bandwidth that is available by a single socket. The small increase used by 4 threads is most likely achieved by effectively using the additional caches, that are not used with two threads. The row-major versions scale better, however provide less performance with a small number of threads. We can see that optimizing the code effectively only gives a small performance increase and as soon as more processing power is added to a socket the gap may close further. However, vector loads can be implemented more efficiently than loads of single elements, so there will most likely always be a performance advantage when using vectors. Nonetheless, when considering the diagonal version requires twice the amount of memory, its performance per used memory is worse than that of the row-major version, even on current systems. The SMP system hardly reacts to the row- and column-based mapping schemes.

Figure 6 shows the performance of the NUMA versions. Both row-major versions scale well and hardly differ in absolute performance. The diagonal version with local memory distribution scales as well as the row-major versions, however the diagonal version with stripe-based distribution does not scale, for the reasons discussed before. Measuring the performance per used memory shows that the row-major versions outperform the diagonal versions. The NUMA system performance indicates that it may scale further with the addition of more CPUs, however doubling the number of CPUs results only in a linear performance increase. The high performance when using matrix sizes up to 4096 is due to the CPU caches, as the NUMA system provides enough cache to store matrixes of this size completely in cache.

V. GPU implementation

GPUs are tile-based many-core systems, which require using tiling in algorithms – e.g. store tiles of data in shared memory – to achieve high performance. Furthermore data locality must be expose among threads in one warp for memory fetches to be coalesced. Both requirements have already been discussed for the CPU implementation, as we have applied tiling to increase cache utilization and memory coalescing matches vector loads and utilization of cache lines / NUMA stripes. So we expect the CPU algorithms to be a good starting point.

We discuss two algorithms to solve the Gauß-Seidel stencil each based on one of the memory layouts. We do not discuss the different mappings of \( V \) to threads, as it does not result in different performance. Our existing CPU algorithm uses barriers and communication between threads, which is not trivial to implement on GPUs. We used a technique called static threadblocks [5] to never-theless implement these. With static threadblocks, a fixed number of threadblocks per multiprocessor is started, so it is guaranteed that all threadblocks are active at the same time. This results in a change of the programming model by promoting threadblocks from data parallel tasks to multiprocessor threads, which behave similar to CPU threads in the sense that they process multiple tasks one after another. Static threadblocks allow for all kinds of synchronization known from CPU programming. Since we have the same synchronization capabilities on both CPUs and GPUs, we implemented similar algorithms. The main difference of the GPU implementations is that we must parallelize the execution within the tile and not only execute multiple tiles in parallel. How intra-tile parallelization is implemented varies, based on the memory layout.

The row-major matrix layout version must at first cope with the problem that accessing diagonals in the matrix is a very ineffective memory access pattern. Directly accessing diagonals in global memory does not allow for any coalescing of memory accesses. We therefore prefetch a tile including a 1-element halo into shared memory, calculate the Gauß-Seidel stencil on this tile, and write it back to global memory afterwards. The maximum number of operations that can be done in parallel when calculating a tile is identical to the width of the tile (not including the halo).
(a) Scalability with different threads. (Matrix size $8192^2$)

(b) Scalability with different matrix sizes with the best performing number of threads.

Figure 5. SMP System

(a) Scalability with different threads. (Matrix size $8192^2$)

(b) Scalability with different matrix sizes with the best performing number of threads.

Figure 6. NUMA System

When prefetching $34 \times 34$ elements into shared memory, only a single warp can actively work on this tile. A block of $34 \times 34$ float elements requires more than 4.5 KB of memory, so we can only have three active threadblocks (and warps) at a GT200 multiprocessor, which is not enough to effectively utilize the hardware. The GF100 multiprocessors provide more shared memory, so we can run 10 threadblocks at the same time. The parallelization within a tile has been designed to rely on the SIMD-lockstep execution of a warp. We directly implement the wavefront pattern within a tile by having only the first thread execute in the first step, the first two in the second step and so on. On average, we have only 16 active processors per threadblock. Increasing the size of the tile without increasing the number of threads per threadblock would increase the number of processors we can utilize in one warp. However with the small amount of shared memory it is hardly possible to increase the size of a tile, as doubling the tile size will (roughly) quarter the number of active threadblocks.

The diagonal-based versions allow to reduce the required shared memory, as diagonals can be read directly from global memory. We must only store three diagonals in shared memory and can thereby increase the threadblock size to 128 (GT200) / 256 (GF100). The memory access pattern is almost identical to the one shown in figure 4, expect that one element per thread are read/written in parallel. We can also utilize prefetching as it has been suggested in [6], loading diagonal $(n+2)$ into registers while computing diagonal $n$. This technique helps to reduce the memory access latency by overlapping memory accesses and calculations. Moreover, the diagonal memory layout allows to utilize all threads except at the beginning and the end of a column.

Figure 7 shows the performance of both algorithms running at a GT200 and a GF100 chip. We optimized the diagonal-based version for large matrices by choosing large threadblocks, which leads to poor work-balance among multiprocessors for small matrices. The diagonal-based version outperforms the row-major version on both GPUs for large
matrices, as it provides a better utilization of the whole system. Nevertheless on the GF100 there is only a small performance difference even though we utilize only 50% of the processors. We can hardly see any scalability in best performance comparing both GPUs, i.e., the best achievable memory bandwidth was hardly increased from one hardware generation to the next. This may indicate that memory bandwidth on GPUs will not continue to rise as rapid as before.

VI. OVERALL PERFORMANCE

The best overall performance is provided by the NUMA system using local memory distribution and SIMD code. Moreover, NUMA systems can easily provide more memory than a single GPU, which is its main advantage. Our NUMA system had 256 GB of memory, whereas current GPUs do not provide more than 6 GB of memory. GPUs fall back in the amount of memory available as they require more expensive memory to sustain the high memory bandwidth. In terms of performance per memory, the diagonal-based code falls behind on all platforms, even on GPUs where we can only utilize half of the processors. Performance increase through parallelism is limited by the wavefront pattern as well as, the tile and matrix sizes. For a matrix size of 8192 and a tile size of 256, we cannot utilize more than 32 cores. On larger GPUs, one may therefore not be able to utilize all MPs. On NUMA systems, it may be feasible to use only a subset of the CPUs for processing, but utilize the memory controllers of the unused CPUs to increase memory bandwidth.

VII. RELATED WORK

Optimizations on stencil codes have been studied intensively, see e.g. Datta et al. [7]. Previous work by the author of the present paper has discussed optimizations of the Gauß-Seidel stencil for multicore CPUs including automatic optimizations [3]. Venkatasubramanian [8] et al. discussed an efficient implementation of the Jacobi stencil for GPUs, which is similar to the Gauß-Seidel stencil, but requires less synchronization.

VIII. CONCLUSION

In this paper, we discussed the Gauß-Seidel stencil on both GPUs and NUMA systems. We choose the Gauß-Seidel stencil as an example for algorithms limited by memory bandwidth and parallelism. We expect more algorithms to fall into this category when future hardware reduces the memory bandwidth per flop and increases the number of processing units. We have shown that it may be feasible to only use a fraction of the processors, if using all processors ends up in a non-optimal usage of e.g. the available memory.

The NUMA system performs best in terms of both performance and performance per used memory, and moreover provides more memory than the GPUs. The latter can easily outperform a single CPU, however currently only provide a relatively small amount of memory.

ACKNOWLEDGMENT

The author thanks NVIDIA for donating the GPUs used for this research. I furthermore thank the management, staff, and facilities of the Intel Manycore Testing Lab for providing access to the NUMA system.

REFERENCES


