FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Professur für Höchstleistungsrechnen · Erlangen National High Performance Computing Center

**Michael Panzlaff**

# Matrix Power Kernels on GPUs

Masterarbeit im Fach Informatik

12. Mai 2024

Please cite as:
Michael Panzlaff, “Matrix Power Kernels on GPUs”, Master’s Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, May 2024.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Professur für Höchstleistungsrechnen
Martensstr. 1 · 91058 Erlangen · Germany

NHR FAU
Erlangen National High Performance Computing Center

# Matrix Power Kernels on GPUs

Masterarbeit im Fach Informatik

vorgelegt von

**Michael Panzlaff**

geb. am 29. Januar 1996
in Backnang

angefertigt am

**Erlangen National High Performance Computing Center**
**Professur für Höchstleistungsrechnen**

**Department Informatik**
**Friedrich-Alexander-Universität Erlangen-Nürnberg**

| | |
|---|---|
| Betreuer: | **Christie Alappat, M.Sc.**<br>**Dominik Ernst, M.Sc.** |
| Betreuender Hochschullehrer: | **Prof. Dr. Gerhard Wellein** |
| Beginn der Arbeit: | **2. November 2023** |
| Abgabe der Arbeit: | **2. Mai 2024** |

## Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

## Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Michael Panzlaff)
Erlangen,
12. Mai 2024

# ABSTRACT

Sparse Matrix-Vector (SpMV) products are used in numerous algorithms of linear algebra and are often the most computationally expensive operation. This includes Krylov Subspace Methods (KSMs) such as GMRES and CG, which rely severely on SpMV to solve linear systems. Since SpMV is known to be very bandwidth intensive, Communication-Avoiding KSMs are an active field of research. They attempt to reduce data interdependencies in the algorithm by replacing expensive SpMV operations with a Matrix Power Kernel (MPK). Doing so allows cache blocking techniques to reuse data from the cache as much as possible in order to reduce the reliance on the limited DRAM bandwidth. However, so far most research has only covered implementations for CPUs. As GPUs are becoming increasingly popular due to their often higher compute and memory performance, we present a benchmark, which runs cache blocked MPKs on GPUs. The benchmark runs on both state-of-the-art Nvidia and AMD GPUs. As a fundamental building block, we use the software library RACE and show that it is possible to get up to a 2× speedup on selected GPUs. Finally, we perform a detailed performance analysis to determine GPU MPK viability for real-world applications. Since a speedup is often not achievable due to kernel call overhead, we predict that MPK performance on future GPUs will heavily depend on their cache size.

# KURZFASSUNG

Dünn besetzte Matrix-Vektor Produkte (SpMV) werden in vielen Algorithmen der linearen Algebra verwendet und sind häufig deren rechenintensivste Operation. Dazu gehören Krylov-Unterraum-Verfahren (KSM) wie GMRES oder CG, welche größtenteils SpMV zum Lösen linearer Gleichungssysteme verwenden. Da SpMV bekanntermaßen sehr bandbreitenintensiv ist, sind kommunikationsvermeidende KSMs ein aktives Forschungsgebiet. Diese versuchen entstehende Datenabhängigkeiten in Algorithmen zu reduzieren, indem SpMV-Operationen durch Matrix-Power-Kernel (MPK) ersetzt werden. Dies erlaubt es, Cache-Blocking-Methoden Daten aus dem Cache so viel wie möglich wiederzuverwenden, damit die begrenzte DRAM-Bandbreite möglichst effizient genutzt wird. Bisher wird jedoch hauptsächlich zu Implementierungen für CPUs geforscht. Da GPUs aufgrund ihrer höheren Rechenleistung und Speicherbandbreite zunehmend verwendet werden, stellen wir einen Benchmark vor, welches mithilfe von Cache-Blocking MPKs auf GPUs berechnet. Dieser Benchmark läuft sowohl auf aktuellen Nvidia als auch AMD GPUs. Die Programmbibliothek RACE wird dafür als grundlegende Komponente verwendet. Wir zeigen auf, dass es damit möglich ist, auf ausgewählten GPUs eine Beschleunigung von 2× zu erreichen. Schlussendlich zeigen wir eine detaillierte Leistungsanalyse um bestimmen zu können, ob MPKs auf GPUs praktikabel sind. Unter anderem wird eine Beschleunigung in vielen Fällen, aufgrund indirekter Kosten, nicht erreicht. Daher prognostizieren wir, dass die MPK-Leistung auf zukünftigen GPUs sehr stark von ihrer Cache Größe abhängen wird.

# CONTENTS

# 1 INTRODUCTION

Solving linear systems is one of the most essential building blocks in scientific computing and there is continuous research and development to make existing solvers more efficient. For sparse matrices it is well known that most iterative solvers are limited by the Sparse Matrix-Vector (SpMV) operation due to its high memory bandwidth requirement. Today, many algorithms operate on Krylov Subspace Methods (KSMs). These are generally considered to be communication[1] intensive as they typically involve a reduction after each SpMV product.

Alleviating the high memory bandwidth requirements is typically done by better cache utilization due to their much higher bandwidth. However having a high amount of communication in the algorithm means that data often has to be fetched from main memory, because it cannot be reused from cache.

When considering problems which involve matrices being larger than the system's cache, the SpMV operation can only be optimized up to the point where it performs as well as the Dynamic Random Access Memory (DRAM)'s bandwidth allows. In order to make cache reuse possible, the SpMV has to be divided into multiple smaller operations which on their own can reuse the data from the cache independently. This type of strategy is also referred to as *cache blocking* as data is processed in smaller blocks and retained as much in cache as possible. However, due to required communication in the solver's algorithm this becomes impossible without making fundamental changes to it.

One of the key elements the authors of [25] use in order to reduce communication and to speed up computation is the use of Matrix Power Kernels (MPKs), which aim to replace the SpMV in the algorithm. These MPKs calculate the span vectors of the Krylov subspace $[x, Ax, A^2x, \cdots, A^kx]$ with the goal to reduce the reduction after each SpMV to a single reduction after each MPK. Of course doing this does not eliminate the communication entirely, as an SpMV in the general sense forms dependencies between its input and output vectors' rows. Since typical matrices are not entirely random the required communication keeps a limited scope and cache blocking can be utilized. An example for that is "CA-GMRES" [25, Sec. 4], an adaption of GMRES, which uses MPKs instead of SpMV. The authors demonstrate that MPKs can be used in algorithms to get speedup of more than a factor of two.

Regardless of whether a linear system solver or Eigenvalue solver, using MPKs in order to gain a speedup is not a drop in replacement for the SpMV kernel. Changes to the algorithms are required. Assuming that the changed algorithms converge equally well, the use of MPKs is however a possible solution for better hardware utilization.

[1] We mean communication in the sense of data dependency and thus the requirement to transfer data in a computer's memory hierarchy. Referenced literature sometimes also uses the term for communication in parallel systems.

One of the main problems that we see is that most of the research focuses on solutions for CPUs. There is some research being done for GPUs [24], but it is rather outdated as it focuses on the case where matrices do not fit in the Graphics Processing Unit (GPU)'s DRAM [24, p. 1750]. These days single GPUs have enough memory to hold the largest possible Compressed Row Storage (CRS) matrices with 32 bit indices and using multiple GPUs is not as essential as in [37]. Because GPUs have become a very fundamental hardware component for scientific computing we benchmark and analyze what level of performance can be expected with the use of MPKs. A core concept our work is based upon is a cache blocking strategy which was originally designed for Central Processing Units (CPUs) [1].

In this thesis we will show the context in regards to related work, discuss the technical fundamentals, and explain how the cache blocking strategy works that we build upon. Using this knowledge we show how we created a cache blocked MPK benchmark for GPUs. We then discuss its optimization and evaluate its performance to show what the challenges and benefits are.

# 2 FUNDAMENTALS

## 2.1 Related Work

The disproportionate increase of memory bandwidth compared to computation speed is no new phenomenon [4, 23, 36]. Accordingly, utilizing the available bandwidth has become more and more crucial. To alleviate this problem, caches have been introduced a long time ago and they usually work well for general purpose applications. However, for larger data sets which do not fit in cache, this usually does not pay off. Therefore *blocked* (or *cache blocked*) algorithms have become an area of research in scientific computing already more than 30 years ago with the goal of avoiding unnecessary or repetitive memory transfers. One of these examples is "The cache performance and optimizations of blocked algorithms" [19]. Its authors take a very fundamental approach at better utilizing cache for dense matrix-matrix multiplications. As for many other cache blocking techniques today, the used core concept is partitioning of the matrix into smaller blocks which fit into cache and consecutively to correctly reorder the processing of these blocks to improve cache reuse.

Their focus is primarily on modelling cache hit and miss ratio depending on the blocking sizes and the cache's parameters, not just including the cache size but also line size and associativity. Unfortunately, the paper's applicability for our research is limited. There is, of course, no roofline based analysis in regard to compute and memory performance, as it has been introduced much later [34]. In addition, compute hardware has vastly changed in favor of multi core computers with highly optimized cache hierarchies. One of the conclusions the authors make is that for a block size "the optimal choice occupies only a small fraction of the cache, typically less than 10%" [19, p. 73]. We assume that this is likely different with today's vastly more efficient hardware.

Albeit older, a more relevant paper is "Implicit and Explicit Optimizations for Stencil Computations" [14]. It discusses the influence of caches on stencil based algorithms like partial differential equation solvers. While they explicitly only discuss stencils, this can still be considered relevant, as stencil operations can be mapped to sparse matrices, which are discussed in this thesis. They present multiple algorithms, starting from a naive implementation, going over to an "implicitly blocked" [14, p. 53] variant to an explicitly blocked algorithm [14, p. 55]. The key concepts they use to improve cache reuse are spatial reuse, which is data reuse within a single stencil sweep and temporal reuse, which is reuse of data across multiple stencil sweeps.

Considering the context of this thesis, there are two reasons why we believe the spatial reuse is less relevant. First, today's compute hardware has changed a lot with better caches and execution pipelines which we expect to be better at hiding instruction latency. We mention this because the authors explicitly discuss SIMD and other techniques as means to hide latency from the compute architecture and we assume this to be negligible for our context. Second, processing sparse matrices is generally even more data intensive than processing stencils, since the matrix data has to be loaded

also. In addition, this matrix data cannot be reused spatially. In contrast, we think that their concept of temporal reuse is a lot more applicable to our goal, especially since a few other papers use similar techniques to speed up subsequent SpMV operations. These papers are reviewed below.

Stencil based solvers are only a subset of sparse matrix based solvers, as stencils are usually implicitly defined by the algorithm instead of a matrix which has to be stored separately. Operating on matrices instead makes it more difficult to generalize what the authors call "time skewing" [14, Fig. 5]. By that they mean a special execution order in which data is accessed that is optimized on cache reuse. Following this execution order is difficult for matrices because they do not necessarily have well structured data dependencies compared to stencils. In contrast, a very recent attempt at a general approach for matrices is done with Recursive Algebraic Coloring Engine (RACE) that we utilize for our work.

However, before discussing RACE, we cover earlier approaches such as done by Demmel et al. [6] which try to apply a blocking strategy for sparse matrix solvers. The authors introduce the concept of Matrix Power Kernels (MPKs) as means to reduce *communication* in KSMs in order to get a modelled speedup in the range of 1.3× to 22.2× [6, Fig. 3, Tab. 3]. At this point an important distinction has to be made as in our thesis we solely discuss local speedup on a single chip. In the cited paper the highest of the performance figures are modelled after a cluster system connected over the internet with much higher latency and bandwidth restrictions. While it does fall under the category of *communication avoiding*, as it reduces memory traffic, that model is not applicable for our case. More comparable speedups are the ones modelled with what they call "CacheBlocked" [6, p. 8] using their *Sequential Algorithms* and they lie in the more moderate range of 1.3× to 2.4× [6, Tab. 3]. Although algorithmically the same as *CacheBlocked*, their "OOC" model is not a good comparison for today's hardware, since it is based on mechanical hard drives. Nowadays these are usually replaced by faster solid state drives and are often irrelevant during the actual computation, as DRAM capacity has become large enough.

Interestingly, their sequential algorithms works similarly to the *time skewing* technique from "Implicit and explicit optimizations for stencil computations", which is referenced previously. In addition there is a further restriction which Demmel et al. have made in regards their sequential algorithms because they only work for sparse matrices that "partition well" [6, p. 1]. This means that the sparse matrices cannot have an arbitrary structure and need to have a grid structure in the graph that represents them. Therefore the applicable problem class is de-facto the same as stencils with the exception that a matrix can have arbitrary weights for each value while a stencil is typically fixed for the entire grid it is applied upon. So this approach is stil not a solution for general sparse matrices.

In a later publication Demmel et al. discuss an MPK alteration of the widely known Generalized Minimal Residual Method (GMRES) [29] algorithm which they call "Communication-Avoiding GMRES (CA-GMRES)" [25, Sec. 4]. Considering the most expensive operations only, they replace the SpMV and the Modified Gram-Schmidt (MGS) from the main loop with an MPK and a TSQR decomposition. They do this similarly to previous ideas [3, 13] of performing GMRES differently, except previous solutions do not use MPKs. Performance measurements are also available and in opposite to the authors' previous papers this one explicitly excludes a "multinode implementation" [25, p. 2]. The speedup for CA-GMRES on their Intel Clovertown system ranges from 1.6× to 4.3× for different types of matrices.

We should note that these improvement figures refer to the entire CA-GMRES, not just the MPK. Our analysis in Chapter 4 will only discuss MPKs and not the performance of linear solvers, so care should be taken when comparing these results. In addition, since the paper was published in 2009, hardware has yet again increased in terms of compute performance, memory, and cache capacity. Thus have the matrices of interests, as the distinction between cached and uncached performance is

of no relevance if the matrices fit in the cache. Nevertheless, when applying this for larger matrices, their conclusion about speedups in regards to rising relative communication costs compared to floating point performance is still reasonable, as this trend has not changed.

A more comprehensive collection of communication avoiding KSMs is published in [12]. They discuss not only the fundamentals of KSMs but also multiple algorithms such as the previous *CA-GMRES* and also CA variants of CG [10] and Lanczos [20]. As their process shows [12, Chpt. 5] adapting an algorithm to avoid communication is by no means trivial and it means that SpMVs are not simply replaced by MPKs for better performance. However, it demonstrates that doing so is possible and that MPKs can be used to speed up real world solvers.

Last we mention more recent research in regards to MPKs and possibilities to replace classic SpMV in linear solvers. The authors of [1] demonstrate the use of the Recursive Algebraic Coloring Engine (RACE) to get speedup for MPKs ranging from 3× to 5× on modern multicore CPUs. As recent chips have seen high increases in L3 cache sizes, this becomes especially relevant since applying cache blocking becomes more effective the more data can be reused. Because the research and analysis of this thesis makes extensive use of RACE we dedicate a separate section to how it works (see Section 2.8).

## 2.2 Outlook

Before discussing the technical fundamentals that we base our work upon we give an outlook on this thesis. We can assume that MPKs play an important role in making future KSM based linear solvers faster, but so far we have discussed very little about computations on GPUs. These days many computations are done on GPUs to take advantage of their higher peak performance and bandwidth. For example, a recent state of the art AMD EPYC 9654P is rated at a maximum nominal memory bandwidth of 460 GB/s [2] while an Nvidia H200 GPU is rated for up to 4800 GB/s [27]. The related research we could find in regards running MPKs on GPUs limits itself to avoiding communication in parallel multi-GPU configurations [37]. However, little to no attention is being focused on sequential execution on single GPUs. Because there is so little research in that regard, we explore the possibilities of performance improvements from MPKs over traditional SpMV on GPUs.

While our detailed analysis will present performance models, one main goal is to perform real measurements on GPUs. We start with a brief summary of GPUs and systems we used. We proceed by creating a standard roofline model based on nominal performance figures for the classic SpMV. Later we will perform measurements with our custom benchmark called *MPK-HIP* and as its name suggests it is written using Heterogeneous Interface for Portability (HIP). This benchmark makes use of common sparse matrix formats for which we also will provide a short overview. As we consider the HIP language to be not so common, we will also provide a brief overview of the language and why we chose it. We continue with a compact introduction into GPU programming and general guidelines for achieving good performance. Last we summarize the functionality of the fundamental building block RACE. This will be relevant for interpreting its performance results later.

## 2.3 Hardware Selection

Recent GPUs have experienced a disproportionate jump in Last Level Cache (LLC) capacity (see Table 2.1). Considering the general increase in computing power for computer hardware, an increase may not be surprising. It is well known that compute performance has increased disproportionately compared to memory performance and thus bandwidth limited algorithms have not benefitted as

much from that increase. Due to this, caches are traditionally used to relieve the memory interface by providing greatly increased bandwidth for smaller data which fits in the cache. This usually occurs implicitly, but the performance may greatly depend on how the programmer hints the processor to perform its memory accesses. As our goal is to utilize the cache more efficiently one can assume that benefits from larger than usual cache sizes are likely.

Before we discuss any of the caching in regards to MPKs, the first step is to choose suitable hardware for benchmarking. As our main goal is cache blocking, we aim at GPUs with higher than usual caches or GPU which are very common. Table 2.1 shows a summary of datacenter and selected consumer GPUs but most of the GPUs disqualify themselves due to their age and small cache. While the AMD MI300X clearly holds the lead with 256 MB, it was unfortunately not available to us at the time. It is followed by the AMD RX 6900 XT with 128 MB, then by the AMD RX 7900 XTX and Nvidia L40 with a slightly smaller 96 MB cache. We chose to use the AMD RX 6900 XT and Nvidia L40 for our testing. The former one due to its high cache size and the latter due to being from a different vendor. As they are a very widespread GPU, final results in Chapter 4 also include results of the Nvidia A100 and H100. Other GPUs are not covered, as we consider their cache size to be too small.

The Nvidia L40 is sold as a data center supported GPU, but the underlying chip is the same as the consumer Nvidia RTX 4090, so we consider this as a de facto consumer GPU. Their main difference being that the Nvidia L40 has fully enabled cache. As it has slightly less LLC compared to the AMD RX 6900 XT we initially did not consider it, but as we later discovered it proved to be a very favorable GPU for MPKs. A complete list of the hardware used can be seen in Table 2.2. For convenience we refer to these systems by their abbreviation.

| GPU | Date | Memory | DRAM type | DRAM bandwidth | LLC size |
|---|---|---|---|---|---|
| Nvidia K40 | 2013 | 12 GiB | GDDR5 | 288 GB/s | 1.5 MiB |
| Nvidia M40 | 2015 | 24 GiB | GDDR5 | 288 GB/s | 3 MiB |
| Nvidia P100 | 2016 | 16 GiB | HBM2 | 732 GB/s | 4 MiB |
| Nvidia V100 | 2017 | 32 GiB | HBM2 | 900 GB/s | 6 MiB |
| Nvidia A100 | 2020 | 80 GiB | HBM2e | 2039 GB/s | 40 MiB |
| Nvidia H100 | 2022 | 80 GiB | HBM3 | 3352 GB/s | 50 MiB |
| Nvidia H200 | 2023 | 141 GiB | HBM3e | 4800 GB/s | 60 MiB |
| AMD MI100 | 2020 | 32 GiB | HBM2 | 1229 GB/s | 8 MiB |
| AMD MI210 | 2021 | 64 GiB | HBM2e | 1638 GB/s | 8 MiB |
| AMD MI250 | 2021 | 128 GiB | HBM2e | 3277 GB/s | 16 MiB |
| AMD MI300X | 2023 | 192 GiB | HBM3 | 5171 GB/s | 256 MiB |
| AMD RX 6900 XT | 2020 | 16 GiB | GDDR6 | 512 GB/s | 128 MiB |
| Nvidia A40 | 2020 | 48 GiB | GDDR6 | 695 GB/s | 6 MiB |
| AMD RX 7900 XTX | 2022 | 24 GiB | GDDR6 | 960 GB/s | 96 MiB |
| Nvidia L40 | 2022 | 48 GiB | GDDR6 | 864 GB/s | 96 MiB |

**Table 2.1** – Memory and last level cache of recent GPUs. The first group shows Nvidia data center GPUs, the second group shows AMD data center GPUs, and the third group shows GPUs with consumer chips.

| Abbrev. | CPU | RAM | GPU | OS |
|---|---|---|---|---|
| RX 6900 | 2x Intel Xeon E5-2620 v4 | 64 GiB | AMD RX 6900 XT | Ubuntu 22.04.04 |
| L40 | 2x AMD EPYC 9354 | 768 GiB | Nvidia L40 | Ubuntu 22.04.04 |
| A100 | 2x AMD EPYC 7713 | 2048 GiB | Nvidia A100 (80 GiB, SXM) | AlmaLinux 8.9 |
| GH200 | Nvidia GH200 | 480 GiB | Nvidia H100 (96 GiB, GH200)) | Ubuntu 22.04.04 |

**Table 2.2** – Test system specifications.

## 2.4 Approximate Roofline Model

Two of the very fundamental performance metrics of a CPU or GPU are their memory bandwidth and floating point performance. An overview of ideal figures for the tested hardware can be seen in Table 2.3.

Using the standard roofline model we can estimate whether MPKs are memory or compute bound. As approximation we use the best-case performance of the SpMV kernel [35, Eq. 4], which for 64 bit floating point values and 32 bit indices results a *data intensity* of 6 Byte/FLOP. For calculating performance, we use the same symbols as [28][p. 77], meaning $I$ for operational intensity, $W$ for FLOPs, $T$ for time, and $Q$ for memory traffic. Additionally, we will also define the inverse of operational intensity as $D = I^{-1}$ for *data intensity* and the bandwidth as $B = QT^{-1}$ for convenience. Thus:

$$P = \frac{W}{T} = \frac{B}{D} \tag{2.1}$$

$$P_{\text{RX6900}} = 512 \frac{\text{GByte}}{\text{s}} \div 6 \frac{\text{Byte}}{\text{FLOP}} \approx 85.3 \frac{\text{GFLOP}}{\text{s}} \tag{2.2}$$

$$P_{\text{L40}} = 864 \frac{\text{GByte}}{\text{s}} \div 6 \frac{\text{Byte}}{\text{FLOP}} \approx 144.0 \frac{\text{GFLOP}}{\text{s}} \tag{2.3}$$

$$P_{\text{A100}} = 1940 \frac{\text{GByte}}{\text{s}} \div 6 \frac{\text{Byte}}{\text{FLOP}} \approx 323.3 \frac{\text{GFLOP}}{\text{s}} \tag{2.4}$$

$$P_{\text{GH200}} = 4000 \frac{\text{GByte}}{\text{s}} \div 6 \frac{\text{Byte}}{\text{FLOP}} \approx 666.6 \frac{\text{GFLOP}}{\text{s}} \tag{2.5}$$

These are approximate figures, yet all the performance numbers are significantly lower than the nominal floating point performance of each GPU. This is the case even when considering that the RX 6900 and L40 have low double precision performance, which is relatively common for consumer chips. According to this basic roofline model, there is a very clear memory bandwidth limit. We discuss more precise figures in Section 4.4.3 as they then depend on the matrix due to row overhead.

| GPU | GFLOP/s (SP) | GFLOPS/s (DP) | GB/s (DRAM) |
|---|---|---|---|
| AMD RX 6900 XT | 23,040 | 1,440 | 512 |
| Nvidia L40 | 90,520 | 1,414 | 864 |
| Nvidia A100 (80 GiB) | 19,490 | 9,746 | 1,940 |
| Nvidia GH200 (96 GiB) | ~67,000 | ~34,000 | ~4,000 |

**Table 2.3** – Nominal GPU performance and bandwidth figures.

## 2.5 Sparse Matrix Formats

### 2.5.1 CRS

For many problems in linear algebra, matrices are only populated sparsely which means that a large percentage of coefficients are zero. For example, a matrix may have millions of rows, but only in the order of ten values per row. If only the non-zero values of this matrix were stored in memory, only approximately 10 million values must be stored. This is easily accomplished with today's computers. In contrast, if the matrix was stored entirely in a dense fashion, trillions of values would have to be stored and processed, which is too much for even a computer with large memories. To overcome this, the Compressed Row Storage (CRS) format is commonly used to store matrices more compactly (see Figure 2.1), which in turn results in more efficient calculations.

The coefficients are stored row-wise in the *values* vector without zeroes. A second vector with the same length called *columns* stores the column index of the original matrix. It is then possible to reconstruct the original matrix by adding a third vector *rowPtr* which stores the index the rows start at in the former two vectors. The length[2] of each row is implicitly determined by the start of the next row.

### 2.5.2 ELLPACK

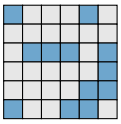
The ELLPACK format can be considered a simplification of CRS as it simply omits the *rowPtr* vector. To be able to reconstruct the original matrix, each row must be equally long since *rowPtr* and *columns* then become a linearized 2D-array. One of the indices is the row's coordinate and the other one is the row index (see Figure 2.2).

Since the resulting data (see Figure 2.2b) is 2D it can be stored row-wise or column-wise, but it typically is stored column wise. Although ELLPACK omits the *rowPtr* vector, it is usually less memory efficient than CRS since all rows have to be padded to the longest row's length. One of its benefits is the option for a column major layout, which improves performance for GPUs, because memory accesses are continuous among threads instead scattered. We discuss these performance implication in more detail in Section 3.2.6.

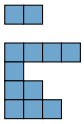
### 2.5.3 SELL-$C$-$\sigma$

In the general case ELLPACK is not a viable option, as it deals very poorly with uneven row lengths. There have been multiple propositions [32, 7, 18] to circumvent ELLPACK's memory and computation overhead. One category of these alterantives are the Sliced ELLPACK (SELL) formats [26, p. 115]. What SELL attempts is to reduce the padding overhead of ELLPACK by slicing the matrix into blocks

[2] By *row length* we refer to the number of non-zeroes per row ($N_{\text{nzr}}$).



**(a)** $6 \times 6$ dense matrix example.

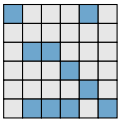


**(b)** $6 \times 6$ CRS matrix example.

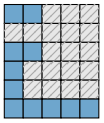
$$
\begin{aligned}
\mathit{values} &= \begin{bmatrix} a_{0,0} & a_{0,4} & a_{2,1} & \cdots & a_{5,3} & a_{5,4} \end{bmatrix} \\
\mathit{columns} &= \begin{bmatrix} 0 & 4 & 1 & \cdots & 3 & 4 \end{bmatrix} \\
\mathit{rowPtr} &= \begin{bmatrix} 0 & 2 & 2 & \cdots & 9 & 12 \end{bmatrix}
\end{aligned}
$$

**(c)** $6 \times 6$ CRS matrix encoding.

**Figure 2.1** – CRS format.



**(a)** $6 \times 6$ dense matrix example.



**(b)** $6 \times 6$ ELLPACK matrix example.

$$values = \begin{bmatrix} a_{0,0} & 0 & a_{2,1} & \cdots & a_{4,4} & 0 \end{bmatrix}$$
$$columns = \begin{bmatrix} 0 & \text{x} & 1 & \cdots & \text{x} & 5 \end{bmatrix}$$
$$N_{\text{nzr}} = 5$$

**(c)** $6 \times 6$ ELLPACK matrix encoding: "x" is a padded column (e.g., zero or previous column).

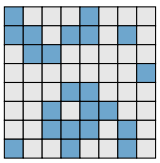
**Figure 2.2** – ELLPACK format.

of rows, which are padded only to the block's maximum row length. A universal and well performing variant is the SELL-$C$-$\sigma$ format [17, p. 9]. It uses fixed size blocks or *chunks* of size $C$ which is chosen to best fit the compute architecture. When $C$ is chosen for example according to the Single Instruction Multiple Data (SIMD) width, the chunk can be read and processed without shuffling. Additionally it specifies a sorting scope $\sigma$ to order the matrix in blocks of length $\sigma$. This is intended to reduce padding in the $C$ sized chunks (see Figure 2.3c). This implies $\sigma > C$ to get any benefits because sorting just inside of a chunk will not reduce the chunk's maximum row length. If no sorting is desired or possible, $\sigma = 1$ is assumed. As we will discuss later in Section 3.2.6, no sorting is done in our examples.

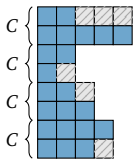
Figure 2.3 shows an example for $C = 2$, $\sigma = 1$ and $\sigma = 8$[3]. For SELL-$C$-$\sigma$ the start of a row can be calculated by first determining the chunk and retrieving its start index via *chunkPtr*. The relative position inside the row can be determined by its internal position just like with regular ELLPACK by using the row's length of its chunk from *chunkLen*.

To conclude, SELL-$C$-$\sigma$ is useful because, unless for extreme cases, it allows to get a good compromise between compactness of CRS and GPU performance from ELLPACK. Accordingly, it is one of the preferred matrix formats we discuss later for benchmarking in Section 3.2.6.

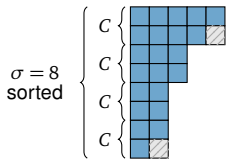
[3]We use the $C$ and $\sigma$ in SELL-$C$-$\sigma$ to specify its parameters: E.g., SELL-$C$-$\sigma$ for $C = 2$ and $\sigma = 8$ would be SELL-2-8.



**(a)** $8 \times 8$ dense matrix example.




**(b)** $8 \times 8$ SELL-2-1 matrix example.




**(c)** $8 \times 8$ SELL-2-8 matrix example.

$$values = \begin{bmatrix} a_{0,0} & a_{0,4} & 0 & \cdots & a_{7,6} & 0 \end{bmatrix}$$
$$columns = \begin{bmatrix} 0 & 4 & \text{x} & \cdots & 6 & \text{x} \end{bmatrix}$$
$$chunkPtr = \begin{bmatrix} 0 & 10 & 14 & 20 & 28 \end{bmatrix}$$
$$chunkLen = \begin{bmatrix} 5 & 2 & 3 & 4 \end{bmatrix}$$

**(d)** $8 \times 8$ SELL-2-1 matrix encoding: "x" is a padded column (e.g., zero or previous column).

$$values = \begin{bmatrix} a_{1,0} & a_{1,1} & a_{1,3} & \cdots & a_{3,7} & 0 \end{bmatrix}$$
$$columns = \begin{bmatrix} 0 & 1 & 3 & \cdots & 7 & \text{x} \end{bmatrix}$$
$$chunkPtr = \begin{bmatrix} 0 & 10 & 16 & 20 & 24 \end{bmatrix}$$
$$chunkLen = \begin{bmatrix} 5 & 3 & 2 & 2 \end{bmatrix}$$

**(e)** $8 \times 8$ SELL-2-8 matrix encoding: "x" is a padded column (e.g., zero or previous column).

**Figure 2.3** – SELL-$C$-$\sigma$ format.

### 2.5.4 Matrix Format Restrictions

Generally, the matrix formats above can use any available primitive data types supported by the compute architecture. Due to implementation restrictions of RACE (see. Section 2.8), we will only consider a single set of data types in this thesis. For floating point values we only use 64 bit double precision and 32 bit integer values for index values. Changing these types may affect performance, as different hardware processes double and single precision floats at different speeds, and the required memory of values affects the needed memory bandwidth.

## 2.6 HIP Language

As of today GPUs are most commonly programmed using the Compute Unified Device Architecture (CUDA). This has the inherent limitation of being limited to Nvidia GPUs, and as we have discussed previously in Section 2.3 we intend to target AMD GPUs as well. HIP [11] is a language which solves this as it is compatible with both AMD and Nvidia GPUs. Unlike OpenCL, porting code from CUDA to HIP is very simple and its API is, with the exception of the *hip* instead of *cuda* prefix, mostly identical. Due to this difference in prefixes, changes to the source code are necessary. For simple programs this can be automated by *hipify.pl* which comes with the ROCm toolchain. The code can then be compiled via *hipcc*, which serves as frontend for *nvcc* (Nvidia) or *clang* (AMD).

In a few cases small adjustments to the code are still necessary. For example, the commonly used *__shfl_XYZ_sync* functions are not available with HIP and have to replaced by their *__shfl_XYZ* equivalent. This is partially caused by architectural differences, because certain AMD GPU differ in warp size and thus cannot use the same function signature.

Once the code works for both vendors, the performance can be expected to be equal to a native CUDA program on Nvidia because HIP's API is simply a header-only wrapper. No additional cost is added, since most function calls from *hipXYZ* forward inline to *cudaXYZ*. As HIP is developed by AMD, we do not expect significant performance penalties from using HIP over OpenCL on AMD. We did not opt for the latest available ROCm version 6.0.0 for Nvidia GPUs, as we were unable to build the HIP API with Nvidia support. More details with regard to software versions are shown in Section 3.2.2.

## 2.7 GPU Programming Introduction

As GPUs have become very widespread and processors have become more parallel instead of higher clocked, the implementation of parallel programs becomes more and more relevant to allow efficient scalability [8]. When a program for CPU is written, hardly any restrictions exist in terms of concurrency. Today's top of the line multi-core processors have 128 cores, but the threads which run on them are usually fairly independent. Consequently each core has its own execution pipeline, a private L1 cache, possibly shared with a single or a very low amount of threads which share the same resources. In contrast GPUs are optimized for processing high amounts of data in a more parallel and regular manner as each "core" usually consists of many threads which run in lockstep. Although CPUs have also gained the ability to process data concurrently on each core via SIMD, on GPUs this is usually implemented via Single Instruction Multiple Threads (SIMT).

Accordingly a number of implications exist. A CPU software library like RACE is written in C++ and cannot run on GPUs. There are of course language extensions like CUDA, which as of today is likely the most widespread language for GPU applications. However, it cannot port software to run on GPU without explicit changes. One of the most fundamental requirements for GPU code are

so-called *kernels*. In principal these kernels are just normal C++ functions with special constraints. For example, only functions annotated with *__device__* are callable from these kernels, and the kernels themselves can only be called via special kernel calls. When such a kernel is called from the CPU, this usually causes very many function instances to be called concurrently. That way all the hardware threads on the GPU can easily be used. If the program only utilized a single or a low amount of threads, performance could be expected to be very poor, as a single GPU hardware thread is typically much slower compared to a CPU hardware thread. Once such a kernel call is issued, the execution usually does not start immediately and is only guaranteed to finish up to an explicit synchronization point (*cudaDeviceSynchronize*). The order of the kernel calls is retained and implicit synchronization happens in between. Subsequent kernel calls thus cannot cause mutual race conditions.

Adapting existing code for CUDA is the first step, but a correct program does not guarantee good performance. The solely reason for porting software to GPU is to get superior performance compared to CPU. Of course general rules for optimization exist for GPUs as they do for CPUs, like reducing algorithmic complexity, reducing code size, avoiding data dependencies, or to avoid branching. But due to their architecture, GPU programs have the following constraints that do not necessarily apply the same way for CPU:

- **Kernel Launch Overhead:** The GPU code is only run via explicit calls from CPU, and for each issued call, CPU to GPU communication is required. In practice, the CUDA runtime can avoid this to a certain extent by batching multiple kernel calls, but even with batching, each new kernel call requires new threads to be launched. It is possible to explicitly batch kernel calls in the CUDA language via *graphs*. That way the driver knows the code and the number of threads ahead of time and can schedule execution more ideally. We will later demonstrate that both number of threads per call (see Section 4.4.2.4) and batching (see Section 3.2.8) can impact the performance.

- **Synchronization Barriers:** While a high number of kernel launches may be limited by the amount of threads the GPU can launch, such a kernel launch always acts as synchronization barrier on the GPU. This provides the programmer a way to avoid race conditions on GPU, but it can cause ramping effects as hardware has to wait for all affected GPU threads to finish before starting new ones. Solving this is similarly affected by solving the kernel launch overhead, as it usually requires minimizing the number of kernel calls. Alternatively, if applicable, fewer synchronizing events have to be used if multiple kernels run concurrently.

- **Uncoalesced Memory Access:** We previously discussed the way hardware threads are organized on GPUs. The smallest group of hardware threads, commonly called *warps*, have a size of 32 on Nvidia GPUs. As these threads typically run in lockstep, all threads will run the same instruction simultaneously. Newer Nvidia GPUs introduced *independent thread scheduling*. Since this doesn't change the general trend how to write good GPU code, it is not covered here. For example, if a multiply instruction is executed, all threads execute this instruction simultaneously. This works well for arithmetic, as it will only interact with data from the local register file. However in case a memory fetch is issued by all 32 threads, the addresses may be arbitrarily scattered in memory. It is well known that random access causes worse performance compared to sequential access, and this applies for GPUs as it does for CPUs. Since the number of running threads on GPUs is much higher than on CPUs this effect becomes significantly more relevant. Threads "prefer" to access memory in a single block, because the memory controller can then for example fetch 32 subsequent *double* values from memory in a single run. If the memory is not accessed sequentially, this may severely impact the performance,

especially on AMD GPUs (see Section 3.2.5). It is also noted that this means data is interleaved for all 32 threads as shown in Figure 2.4 and not ordered sequentially for each thread.

- **Code Branching:** As for the previous point, similar restrictions apply for conditional branches. When optimizing code for CPUs, code branching can be reduced to avoid mispredicted, and thus penalized, branches. On GPUs this is usually problematic due to the lack of branch prediction. Instead, the reduction of diverging branches is more relevant. Once a thread branches to a different target than the ones in the same warp, it can possibly stall all other threads within that warp.
- **CPU-GPU data exchange:** Because the compute work for GPU is dispatched from CPU, the data set, in our case a matrix and a vector, has to be transferred from CPU to GPU before running the kernel. Accordingly the results have to be transferred back to the CPU. This imposes a possible transfer bandwidth limitation and a synchronization barrier. This is however only relevant for the start and end of computation. If all computation is done on GPU, no data exchange is necessary between kernel calls. For problems involving many iterations this can usually be ignored.

Except the last of point, all will be relevant for the following chapters. We will reference them accordingly for certain design decisions we made.

## 2.8 RACE and Level-Based Cache Blocking

The proposed solution to GPU accelerated MPKs is primarily based on the Recursive Algebraic Coloring Engine (RACE) [1]. RACE is a software library which implements a temporal blocking algorithm. This is used to better utilize the cache during MPK calculation. This section briefly explains how RACE works and how it creates the fundamental building block for the later used benchmark implementation called *MPK-HIP*.

One of RACE's first steps is to apply a transformation to the matrix to make it more diagonal and thus more "cache-friendly" [1, p. 586]. This transformation, which is solely a row and column

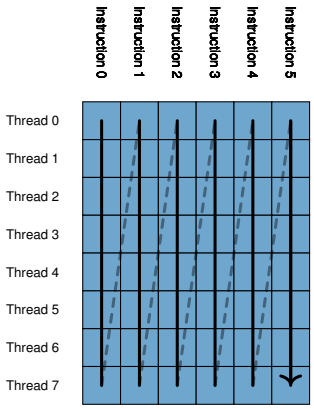


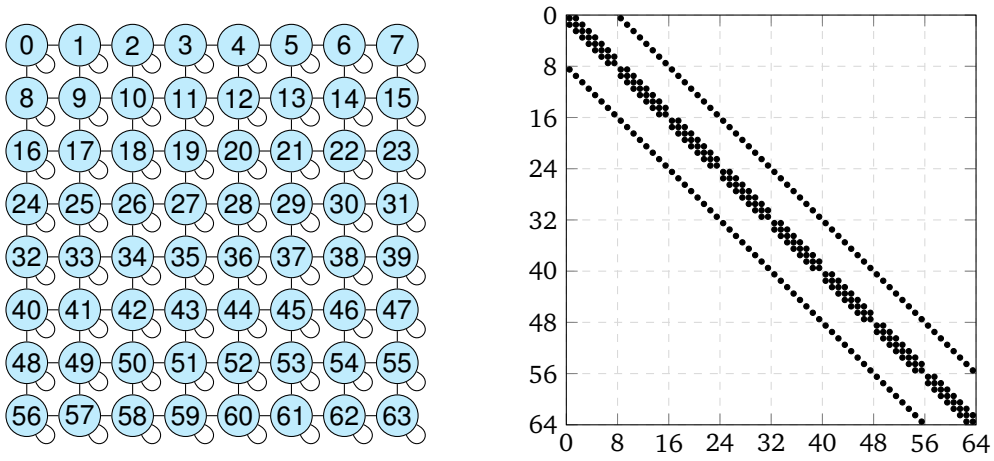

**Figure 2.4** – Memory Coalescing: The black arrow denotes the ideal data ordering for memory read/write coalescing ($N_{\text{tpw}} = 8$). As all threads in a warp execute the same instruction at the same time, they should operate on a continuous block of data for every memory operation.

permutation, is non-optional and is also required for the implementation of *MPK-HIP*. Matrix permutations are commonly used to speed up SpMV [15], and we will assume that its gains will generally outweigh its costs. As RACE uses an algebraic graph-based approach, the input matrix does not require any specific shape [1, p. 581], but of course the gains are limited to the possibilities of permutations and cannot fix a random sparse matrix.

For better illustrating RACE's exact functionality, we use a simplified 2D-5pt stencil matrix instead of a 2D-7pt stencil as originally published in [1, p. 586] in an $8 \times 8$ grid. This stencil's matrix is shown in Figure 2.5b along with its graph[4] in Figure 2.5a. With the exception of the edge cases, the matrix has a diagonal shape. Because calculating matrix powers involve successive SpMV operations, these SpMV operations can be characterized by row dependencies. This is meant in the sense, that each output vector component for an SpMV operation depends on a set of input vector components, which can then be represented as a graph. In this graph, each vertex represents a row, and an edge forms a dependency of a row with another row. The corresponding example graph is shown in Figure 2.5a. Since a 2D-5pt stencil is used, each row's dependencies are its vertical and horizontal neighbours but also the row itself.

This graph becomes relevant when attempting cache blocking, because it restricts which data may be retained in cache, and which data is required to be fetched from memory. In order to calculate one power iteration for a certain row, the graph has to be traversed from that row's vertex by a depth of one in order to determine which input rows are required. For cache blocking it is necessary that these inputs rows stay in cache as much as possible, but for an arbitrary matrix power this is impossible. In that case the graph will be traversed completely at some power. Thus all the rows must be kept in cache, which is not possible. RACE attempts to solve this limitation by a calculation order strategy which ensures that data fetched from memory will stay in the cache as long as it still needed for calculation. A matrix that already fits in the cache would of course not benefit either way.

[4]We use the term *graph* or *dependency graph* as means for the equivalent matrix interpreted as adjacency matrix.




**(a)** Row dependency graph.

**(b)** 5pt-stencil 2D matrix.

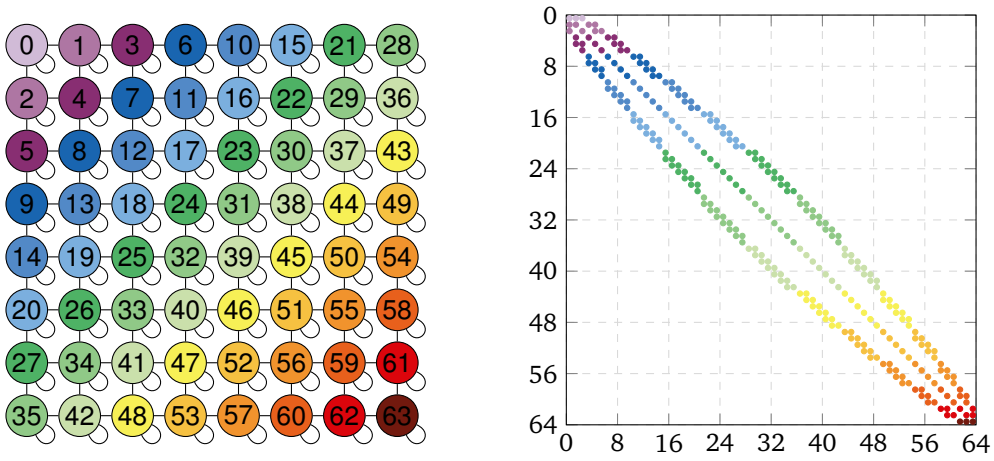
**Figure 2.5** – 2D 5pt-stencil matrix structure and dependency graph. A vertex represents a row of the matrix. An edge denotes a dependency and points to another row which is needed for the other row's calculation [1, Fig. 4].

While the previously mentioned permutation step does improve cache-friendlyness it is also fundamental to RACE's "level formation" [1, p. 586]. These levels are key to define the calculation order of the final matrix power. In this formation step rows are assigned to a level number in such a way that all rows in a level only depend on other rows of neighbouring levels. The way the level numbers are assigned is a traditional Breadth-First Search (BFS) where the depth of the search defines the level number, and the search order of the rows defines their new row. Thus each depth descent in a BFS searches the required dependencies for one additional power iteration.

The level formation and the reordering is shown in Figure 2.6a. All vertices now have a different number assigned which refers to the new row they have been reordered to. As expected, the numbers occur in the BFS order from the starting row zero. Each BFS descent is shown as new level in a separate color and as reordered matrix in Figure 2.6b. The graph layout is unchanged, as dependencies are neither introduced nor removed.

We assume that once an SpMV is executed on a single level, its output will stay in cache. If $p = 1$ is calculated all inputs will not be in cache. One can then use an execution order illustrated in Figure 2.7 (black arrows) to maximize cache reuse. Because all rows in a level only depend on rows of their neighboring levels, a level's power $p + 1$ can be calculated fully cached as soon as its neighbouring levels' power $p$ are calculated (pink arrows). Each square represents an SpMV operation to calculate a level's power. If the execution is performed in the demonstrated order, at most $p + 1$ levels will have to stay in cache. In theory only $p$ levels have to stay in cache due to the diagonal traversal, but because caches often employ a LRU eviction policy, it is $p + 1$ in practice. With this order a level is fetched from DRAM only for its first power calculation. This is similar to the sequential algorithms for communication avoiding KSMs that we discussed previously in Section 2.1.

Because level size can be very small and possibly very unequal in size, RACE forms *level groups* as an additional optimization step. The approach is to reduce execution overhead by being able to perform larger continuous SpMV operations by merging multiple levels into one. There is generally no downside to forming as large level groups as possible, but eventually an increase will hit a cache size boundary where $p + 1$ level groups no longer fit in the cache. This boundary is also mentioned




**(a)** Row dependency graph.

**(b)** 5pt-stencil 2D matrix.

**Figure 2.6** – 2D 5pt-stencil matrix structure and dependency graph after BFS reordering matrix from Figure 2.5. Each color represents an assigned level and each level's rows only reference rows from themselves or neighbouring levels [1, Fig. 4].

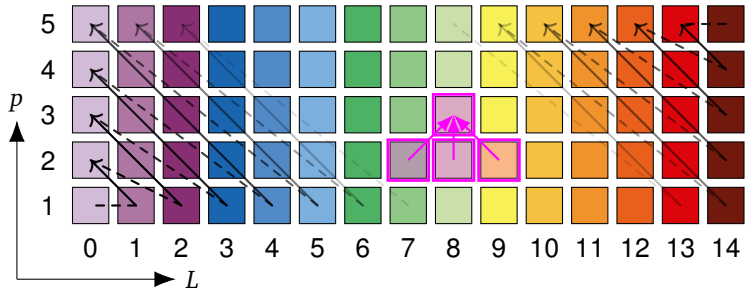



**Figure 2.7** – $Lp$ diagram: Black arrows denotes the most cache efficient execution order. Each square represents an SpMV performed on the respective level [1, Fig. 5]. When following this order, at most $p+1=6$ levels have to be retained in cache in order to avoid DRAM loads for $p \neq 1$. If for example $L_{p=5}(0)$ is calculated that level is no longer needed, but $L_{p=1}(5)$ will not evict L(0) from cache due to being used just before. This is why $p+1$ levels need to fit in cache instead of $p$. The pink arrows show which data level $L=8, p=3$ needs for its calculation. The diagram applies for levels and level groups equally.

in [1, Eq. 5] and accounts for the non-zeroes in the matrix data. [1] does not mention result vectors explicitly, but it is rather simplified in a safety factor which accounts for all other data overhead as well. We will discuss this safety factor later in Section 4.4.2.5 as it can have an effect on the cache blocking efficiency.

In contrast to being small, a level can also be too large. That is the case if $p+1$ levels of that size can no longer fit in cache. RACE supports a way to split larger levels into smaller ones using "Recursion" [1, p. 590]. However due performance penalties of more fine grained execution flow (see Section 4.4.2.4), we have not analyzed it in our work.

When the programmer utilizes RACE the level and group forming is abstracted away. Consequently only the reordering has to be performed as supplied by RACE, and and SpMV kernel must be provided. Because RACE has to be able to perform SpMV on individually formed level groups, not all SpMV kernels can be used since they usually do not support execution on a subset of rows. This unfortunately includes commonly used math libraries such as *cuSPARSE*. That is why we chose for a custom kernel implementation, which we cover in more detail in the following chapter.

# 3 IMPLEMENTATION

## 3.1 Benchmarking Goals

The main objective we are aiming for is to reliably evaluate performance of MPKs on different hardware configurations. We already mentioned briefly in Chapter 1, the core of this thesis is the *MPK-HIP* benchmark. Before discussing its functionality in the following subsections it is important to elaborate the actual goals of this benchmark and its results.

Fundamentally, a benchmark is supposed to allow "evaluation of the speed of the compilation and execution" [22, p. 83] of software. The general intention is to analyze our improvements to linear systems solvers. It is commonly known that SpMV operations, or in our case MPKs, are the most limiting factor. Thus we can limit our benchmark to this particular matrix power *kernel*. In addition to just running our own custom kernel, it is pivotal to judge whether it runs at the speed it is expected to. For example, the results should follow the *roofline* model [33], alternatively it should be understood why the benchmark does not perform as expected compared to the roofline. Unless optimized MPKs vastly change the compute-to-bandwidth ratio compared to regular SpMV, they are expected to be severely bandwidth limited. In our case this is likely not the case, as RACE does not alter the total amount of computation nor reduces the amount of data compared to regular SpMV based MPKs. However, we can still expect a speedup with RACE, as caching will raise the bandwidth limit effectively. Running into a compute limit will not be a problem. If we recall Table 2.3, this would require speedups over 10×, which would be very optimistic.

## 3.2 MPK-HIP

### 3.2.1 Benchmark Build Setup

In this section we present our benchmark *MPK-HIP*, explain how it works, and what it measures. We briefly mentioned the CUDA language as means to program GPUs. As one of our test systems features an AMD GPU, CUDA cannot be used. For older GPU generations OpenCL used to be the common choice for AMD. However since OpenCL is known to run worse on Nvidia GPUs [16], and to ease porting, AMD ROCm introduced HIP. This created an easier way to run the same code for both Nvidia and AMD GPUs. We chose HIP as language in order to run the same benchmark on both vendors' GPU to avoid relying on multiple benchmark codebases, which could possibly diverge.

Many benchmarks traditionally have a very simple build setup. Commonly just a single Makefile and a single source file. GNU Makefiles can of course be used with multiple source files, but the HIP language leads to a non trivial setup, as manually setting up vendor specific compiler paths

and include directories becomes rather cumbersome. In addition, MPK-HIP also depends on the RACE software library, which uses CMake to manage its builds. As a single source file benchmark is already out of question, and as MPK-HIP also relies on other libraries, using CMake is inherent. While CMake does make managing builds easier, it has the limitation that it makes it harder to run the benchmark on numerous platforms. This is not a limitation in our case, as running code on GPU already vastly reduces the number of hardware architectures. As of today this leads to a modern x86 oder ARM based system with either an Nvidia or AMD GPU. It is unlikely that CMake is not available for these, especially considering CMake is recommended by AMD to be used for the HIP language. One of their reasons to recommend CMake is because it natively supports the HIP language and it can compile for both Nvidia and AMD via switch. It does however require the very recent version 3.28 of CMake.

Another practice many benchmarks follow is to ship certain libraries they depend upon. Common examples are argument parsers or matrix file parsers. We excluded this approach, as CMake already provides a way to fetch and find required libraries via *FetchContent* and *find_package*. Consequently it keeps the actual benchmark free of unrelated code. This approach may lead to the benchmark being more prone to build failures, as it depends on libraries which may not be as easily obtainable in the future. However, since the ROCm libraries are non-trivial to setup and cannot be included in the project in the first place, this becomes comparatively less of a problem.

In the end, compiling MPK-HIP is as simple as compiling any other CMake based project. The user runs `cmake ..` in the build subdirectory and optionally specifies `-DMPKHIP_GPU_ARCH` to select which GPU architecture to compile for. Alternatively, the included `build.sh` simplifies the GPU architecture choise by providing an interactive selection.

### 3.2.2 Dependencies

We will briefly discuss the software libraries MPK-HIP depends on. Some of them provide merely "boilerplate" code for features which C++ natively does not support, like JSON (via *nlohmann::json*), argument parsing (via *argparse*), or modern format strings (via *libfmt*). Others provide more important functionality:

- **Eigen:** The original RACE CPU benchmark uses custom implementations for dense and sparse matrices. One of the reasons is that the CPU benchmark requires NUMA aware memory allocation to achieve best performance. Since our benchmark runs on a single GPU, Non-Uniform Memory Access (NUMA) nodes are not relevant for kernel execution and no custom allocation is required. Due to this, we chose to use an off-the-shelf solution for matrix handling on CPU before it is uploaded to GPU. In addition we used it to provide a CPU-based reference SpMV implementation for MPK to detect implementation errors.

- **fast_matrix_market:** Because MPK-HIP has to ultimately load matrices from disk for benchmarking, a solution for matrix loading is needed. In this chapter we only present results for custom generated stencil matrices, so in theory a custom solution would be sufficient. As other matrices from the SuiteSparse Matrix Collection [31] are discussed in Chapter 4, we chose to use the Matrix Market format. Many existing benchmarks come with the NIST implementation in ANSI C or use a custom parser. We excluded it, as the NIST implementation requires extensive changes to correctly interoperate with modern C++. We also chose against an entirely custom implementation as they often do not support all types of the Matrix Market format. *fast_matrix_market* addresses that and in addition it can natively parse matrices into Eigen matrix objects.

- **Thrust:** As MPK-HIP attempts to use modern C++ coding standards, RAII memory management is preferred over manual memory management. For GPU this is achieved by using Thrust's *device_vector* instead of manual handling via *hipMalloc* and *hipFree*. In addition, it provides type checking for all memory handling outside of the GPU kernels.

A complete list of dependencies and versions is given in Table 3.1.

### 3.2.3 Operation Procedure

In the following paragraph MPK-HIP's benchmark procedure is shown. The RACE source code comes with example benchmarks, in particular *mtxPower*, which our benchmark is based upon.

1. **RACE permutation:** The matrix file is loaded from disk and RACE performs its matrix permutation as explained in Section 2.8.

2. **Instantiate Kernel:** Since different types of kernels are implemented, different matrix formats are used on GPU depending on the kernel the user selects. These formats are explained in more detail in Section 3.2.5 and the following sections. It is possible to choose between a fixed-value filled vector or randomly filled vector for matrix power initialization.

3. **Warmup:** As the very initial run may exhibit poor performance due to cold caches and unusually high overhead, the kernel is executed a few times. In addition this step checks how many times the kernel has to be called in order to run for a single second. This measurement is performed several times while increasing the call count, since a very low number of kernel calls may still have very high overhead. By increasing the call count that initial overhead is reduced to a minimum.

4. **Validation Run (optional):** When implementing a new kernel or determining the accuracy of an existing kernel, it is important to compare the results against a reference implementation. In this step a CPU based implementation is performed via a simple matrix-vector product from the Eigen library. This allows to detect numerical or implementation errors.

5. **RACE Kernel Run:** Since the number of iterations has already been determined during warmup, the kernel is now called that number of times. Either it is called directly, or by first recording a HIP graph and then executing this graph (see Section 3.2.8). By running the kernel, we

| Library | Description | Version |
|---|---|---|
| RACE | MPK cache blocking library | latest git |
| ROCm | HIP language and AMD GPU support | 5.7.1 / 6.0.0 |
| CUDA | Nvidia GPU support | 12.2 |
| Thrust | GPU memory management | (see CUDA/ROCm) |
| fast_matrix_market | Matrix Market reader/writer | 1.7.5 |
| Eigen | Linear Algebra Math Library | 3.3.8 |
| argparse | argument parsing | 3.0 |
| libfmt | modern C++ string formatting | 10.1.1 |
| nlohmann::json | JSON library | 3.11.2 |

**Table 3.1** – List of MPK-HIP's dependencies. ROCm 6.0.0 was only used for AMD GPUs. RACE does not have stable version numbers.

actually refer to instructing RACE to execute the kernel with enabled blocking in order to calculate the requested power. Explicit synchronization barriers (*hipDeviceSynchronize*) are used before the first and after the last iteration to solely measure the time spent on actual kernel execution. Memory copies from or to the GPU are not included.

6. **Result Comparison (optional):** If validation was requested previously, the reference implementation is compared to a single kernel run via RACE. The accuracy results are logged accordingly.

7. **SpMV Kernel Run:** The purpose of MPK-HIP is to determine whether it is possible to get better performance via RACE. In this last step the same power is calculated with the same number of runs to determine the speedup of RACE compared to an implementation via traditional matrix-vector products. This classic SpMV run uses the already permuted matrix in order to reduce the speedup to effects from cache blocking only.

If no validation is requested, the benchmark will run and output its values as JSON. This allows processing the results via scripting languages. Our exact procedure is explained in Section 3.3.

Unfortunately we discovered a problem with the RACE library where it would ignore the number of threads passed during initialization. When running the benchmark, care must be taken to set the environment variable `OMP_NUM_THREADS=1`. Otherwise RACE may malfunction and cause MPK-HIP to report incorrect results.

### 3.2.4 Implementation and Optimization

In the following sections we explain the functionality of the most important kernels that we implemented. The same kernel is used to benchmark cache blocked MPK performance with RACE and classic SpMV MPK. Thus optimizations to the kernel will speed up both RACE and SpMV based MPKs.

During optimization we benchmarked these kernels against two standard matrices: A 3D 7pt 256x256x256 stencil matrix and a 3D 27pt 128x128x256 stencil matrix. We chose these as they typically perform better due to their regular structure. The 27pt stencil grid size was decreased to offset for the larger stencil compared to the 7pt. Both matrices do not fit in the cache of any GPU we tested (see Table 4.3).

The code of the kernels that are shown in the following sections are simplified versions only. They do not cover edge cases of HIP block sizes and do not show arbitrary row start and end pointers. Because these cases only slightly change the row index calculation and early kernel returns, we do not consider these relevant in terms of performance.

### 3.2.5 CRS

In this chapter we discuss the first fundamental kernel, which is MPK based on the CRS[5] format. It is very similar to the CRS kernel used in *mtxPower* from RACE's original source code and serves as the first level to compare performance of moving from CPU to GPU. For MPK-HIP and mtxPower, there is no inherent requirement to use the CRS format. This only refers to the calculations since during preprocessing RACE still needs to the matrix in CRS format. For the kernel, the only requirement is that it uses a row based format, which means that rows need support for random access. While the rows are not necessarily processed in a random order, RACE requires its kernel to start and to

[5] In practice the terms CRS and CSR are often used interchangeably. As following plots may hint, the MPK-HIP kernel implementations use the term CSR internally.

stop at an arbitrary row when processing a level group. Accordingly, we will also cover other matrix formats than CRS in the following sections.

The basic functionality of the standard CRS kernel and its memory layout is shown in Figure 3.1. In contrast to performing a matrix-vector product on CPU, where one thread typically processes a large number of rows, the high thread count on GPU requires the work to be split up into smaller pieces. Our implementation assigns a single GPU thread to a single row. If the matrix has a sufficient number of rows, the GPU will be saturated. For example, on the RX 6900 that would be 81920 rows and 218112 rows on the L40. In opposite, if the number of rows, thus threads, is too small, the GPU will not be saturated and performance may not be ideal. However, we do not optimise for small matrices because cache blocking is only meaningful solution for data which does not fit in the cache.

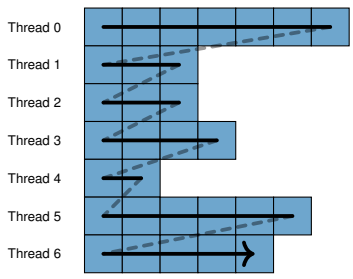
Figure 3.2 shows the performance of the naive GPU CRS kernel plotted over a range of powers. As mentioned previously, two stencil matrices are shown, and their MPK performance is compared between RACE and a regular SpMV based MPK. A very clear problem can be seen in the graphs with regard of performance on the RX 6900. Not only does the RX 6900 not benefit at all from RACE, it actually is worse than a traditional SpMV MPK calculation. Additionally, the figures are much lower than the approximated 85 GFLOP/s from (2.2) for regular SpMV. In contrast, the L40 becomes three times as fast as regular SpMV using RACE power calculation. However, no false conclusions should be made about speedup as long as the SpMV based MPK runs suboptimally. Once one considers that both cards exhibit only half the performance on the 27pt stencil with SpMV, it likely runs very poorly. Furthermore it is also far below the 144 GFLOP/s roofline.

These roofline figures ignore vector overhead and assume ideal caching behavior for `col[i]`. Accordingly the data intensity of 6 bytes per FLOP are approximate only. For very sparse matrices this can be 9 bytes or higher, but for the two stencils matrices it is in the range of approximately 6 to 7. Therefore estimation errors within this relative deviations are tolerable, but as shown the measured performance is much larger, especially on the RX 6900.

One common problem on GPUs for CRS kernels is load imbalance, as each thread may have to process a vastly different number of coefficients. This is not investigated, as stencil matrices do not have this problem since most rows are equally long. Both matrices under test are larger than one gigabyte to make sure they do not fit in the cache of the tested GPUs. RACE's cache size parameter is chosen with twice the size of the GPU's LLC. This is due to an implementation detail of RACE and is discussed in more detail in Section 4.4.2.5, but it also does not affect the standard SpMV results. As load imbalance is likely not the problem, the poor numbers must be caused by something else.

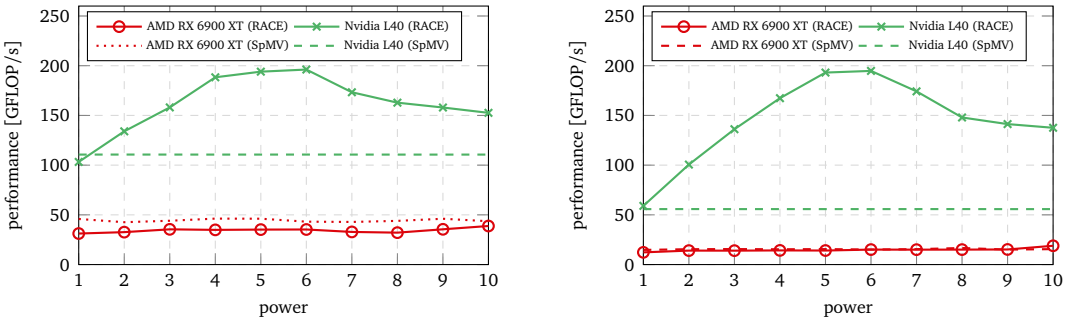
```
1 void gpu_crs(val, col, rowptr, b, x) {
2     row = blockIdx.x * blockDim.x + ↘
          threadIdx.x;
3     sum = 0.0;
4
5     for (i = rowptr[row]; i < ↘
          rowptr[row+1]; i += 1)
6         sum += val[i] * x[col[i]];
7
8     b[row] = sum;
9 }
```

**(a)** CRS kernel (*gpu-csr*).




**(b)** CRS data order.

**Figure 3.1** – CRS on GPU example: The black arrow shows the order of the non-zeroes in memory.




**(a)** 3D 7pt-stencil (256x256x256).

**(b)** 3D 27pt stencil (128x128x256).

**Figure 3.2** – MPK-HIP: *gpu-csr*.

Unfortunately it is not possible to simply use vendor-optimized kernels, as they are typically limited to operate on entire matrices and not just slices of rows. A key element that we have discussed in Section 2.7 is the correct coalescing of memory accesses. In the following section we will explain why this is a huge problem with the CRS kernel.

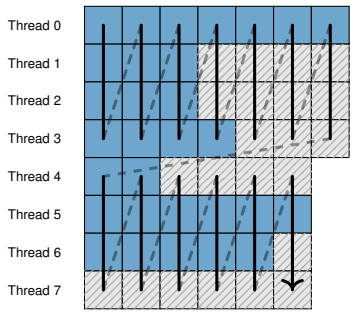
### 3.2.6 SELL-$C$-$\sigma$ (naive)

The CRS format does a reasonable job to keep the sparse matrix' memory footprint compact. As we briefly mentioned, good memory ordering is crucial to get good performance on GPUs. Unfortunately CRS is not ideal in this case. Good memory ordering means that all threads in a warp access the memory in a continuous block. Although potentially an extreme example, Figure 3.1 shows that the first column processed, which is accessed simultaneously, does not lie continuously in memory as the arrow indicates. When all threads fetch a coefficient from the first column simultaneously, the block accessed is not continuous and cannot be coalesced. Multiple matrix formats attempt to circumvent this, and most of them use a column-major layout instead of a row-major layout.

One of the challenges that a column-major layout has to solve is how to deal with uneven row lengths. All coefficients in neighbouring rows are supposed to be processed in parallel and the

```
1  void gpu_scs(val, col, chunkPtr, chunkLen, b, x) {
2      row    = blockIdx.x * blockDim.x + threadIdx.x;
3      chunk  = row / C;
4      idx    = row % C;
5      ptr    = chunkPtr[chunk];
6      rowlen = chunkLen[chunk];
7      sum    = 0.0;
8
9      for (i = 0; i < rowlen; i += 1) {
10         valColIdx = ptr + i * C + idx;
11         sum += val[valColIdx] * x[col[valColIdx]];
12     }
13
14     b[row] = sum;
15 }
```

**(a)** SELL-$C$-$\sigma$ kernel (*gpu-scs-cm-trad*).




**(b)** SELL-4-1 data order, correct memory coalescing for $N_{\text{tpw}} = 4$, stripe shows padding.

**Figure 3.3** – SELL-$C$-$\sigma$ on GPU example: The black arrow shows the order of the non-zeroes in memory.

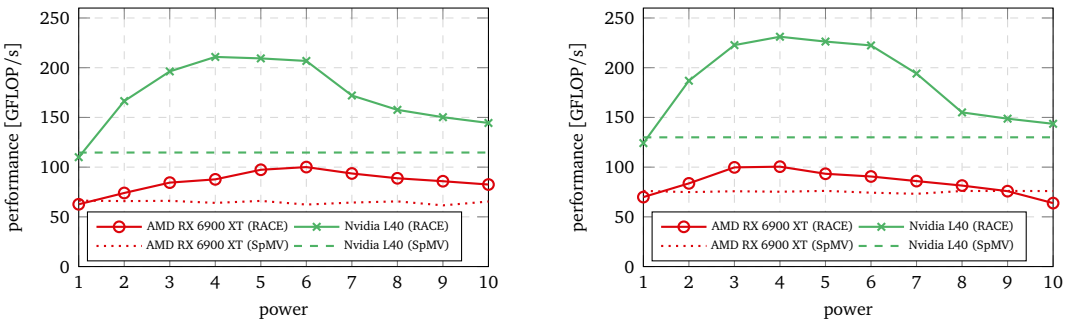
uneven memory accessed has to be fixed. This is usually solved by padding rows to be equally long, and a simple example of this is the ELLPACK format, which we covered previously. Although not with a stencil matrix, the ELLPACK format has the disadvantage that very few long rows can greatly increase matrix size. MPK-HIP implements an ELLPACK kernel, but we do not discuss its results, as we consider SELL-$C$-$\sigma$ to be a better compromise for arbitrary matrices.

With SELL-$C$-$\sigma$ the matrix is only ordered column-major for small chunks with $C$ number of rows (see Figure 3.3b). If the chunk size $C$ is chosen to fit the GPU's warp size of 32, memory access can be coalesced and thus better performance is expected. One of the reasons why SELL-$C$-$\sigma$ can reduce the amount of padding is that the rows are only padded to the maximum row size within a chunk. This can be improved further by increasing the sorting context $\sigma$ since large rows will be grouped with other large rows. Unfortunately it is not possible to reorder rows in our case because this will conflict with RACE, as it already reorders the matrix during level formation. While this makes it impossible using SELL-$C$-$\sigma$ to its full extent, it is still better to use it with no sorting instead of using CRS or ELLPACK. Therefore MPK-HIP implements SELL-$C$-$\sigma$, but only with $\sigma = 1$. This includes all results in this chapter and the final evaluation (see Chapter 4).

In Figure 3.3b the results for the SELL-$C$-$\sigma$ kernel can be seen. Compared to previous results in Figure 3.1, the graph for the RX 6900 is vastly different. The baseline performance with the SpMV 27pt kernel speeds up from 15 GFLOP/s to 75 GFLOP/s. This is already much closer to the 85 GFLOP/s roofline. Although not as high, the SpMV performance for the 7pt kernel also increases. It is an extreme example, but it shows very clearly why coalescing memory accesses is important to achieve good performance on GPU.

Because the performance of the L40 with CRS was already much better, its difference is not as large. For 27pt SpMV performance increases by at least a factor of two while the 7pt performance remains for the most part unchanged. The L40 already experienced speedup over SpMV with RACE with CRS, but in this measurement with SELL-$C$-$\sigma$ the peak performance is slightly higher while performing a lot better overall for non ideal powers. Overall, using SELL-$C$-$\sigma$ over CRS is a very clear decision. Unfortunately the RX 6900 does not profit from RACE as much as the L40, but more detailed explanations for this behavior can be found in Chapter 4.

With this higher base performance, one more effect is visible in the RX 6900 graph. SpMV performance is expected to be constant regardless of the power calculated, but the RX 6900 graph is not entirely flat. There is a peak-to-peak variation of about 5% for all different powers. The graph does not show that this 5% variation also occurs on a run to run basis with the same power. Consequently this can be attributed to measurement errors. It is unclear whether this is caused by




**(a)** 3D 7pt-stencil (256x256x256). **(b)** 3D 27pt-stencil (128x128x256).

**Figure 3.4** – MPK-HIP: *gpu-scs-cm-trad* ($C = 32, \sigma = 1$).

the GPU hardware or the driver stack, but due to HIP's novelty, we consider a driver problem to be a reasonable assumption. Since the L40 does not show this problem in this graph nor in any other graph that we present, we assume that the benchmark itself can be excluded as root cause.

The 7pt stencil performance is overall lower compared to 27pt, but that is expected, as with fewer values per row cause more overhead per row. Exact explanations for these differences can be found later in Chapter 4. Yet there are still more options for optimization. During execution we observed for the RX 6900 and the 27pt matrix that each RACE kernel call would operate on ∼47000 rows. In contrast, each classic SpMV kernel would operate on ∼4200000 rows. The latter one easily saturates the RX 6900 with threads while the former one does not. We considered that this could have a negative impact on achievable memory bandwidth.

With the currently described SELL-$C$-$\sigma$ kernel, a single thread is assigned to each row like for CRS. In order to exclude the possibility of the GPU not being saturated properly, we opted to extend this SELL-$C$-$\sigma$ kernel to use more threads per row. This is covered in the next section.

### 3.2.7 SELL-$C$-$\sigma$ (multi threaded)

In this section we show how to extend our previous SELL-$C$-$\sigma$ kernel to utilize multiple threads per row. Its main intention is to increase the total number of threads in order to saturate the GPU, which would not saturate otherwise. We will assume that the matrix benefits from that, even though there may be cases where using more threads may result in worse performance. For example, if the number of rows is already high enough to launch enough threads, this optimization is unnecessary and may even be counterproductive.

The simplest solution to use multiple threads per row is to first divide a row by the number of threads it is supposed to be processed by and let second each thread only multiply and add a subset of the row. The rows are then divided by a thread count which is a power of two, as this simplifies reduction. That way each row's result can be accumulated like a binary tree. Dividing by a power of two has the distinct disadvantage of more idling threads, as a row length may not necessarily be divisible by a power of two. This is a consciously accepted problem, as the only well performing solution would be to pad the matrices to the correct row length. However, padding the rows to a length of a power of two would increase storage requirements beyond SELL-$C$-$\sigma$'s usual padding. In that case the additional bandwidth required could substantially decrease speedup.

During implementation, we considered multiple methods of reduction. One of the most commonly used methods to interchange data with nearby threads are *shfl* instructions, albeit being limited to threads from within the same warp. This is generally the fastest way for data exchange between threads, as there are little to no synchronization cost. On the other hand it makes it impossible to exchange data with threads from different warps which limits the maximum usable number of threads per row to 32[6]. So using more threads requires different synchronization and interchange mechanisms, shared memory being one example. Multiples warps can access shared memory from within a *block* which would increase the possible number of threads to typically 1024[7]. We initially attempted to implement a solution using shared memory. Since multiple warps do not run interlocked they require explicit synchronization, and therefore the performance deteriorated by a factor of two. Thus we did not continue to analyze this approach.

Due to the former reasons, our multi threaded kernel uses a *shfl* instruction as means for result reduction. This restricts the kernel to a thread increase factor of 32. One could construct sparse matrices where that factor still would not launch a sufficient number of threads. However we assume

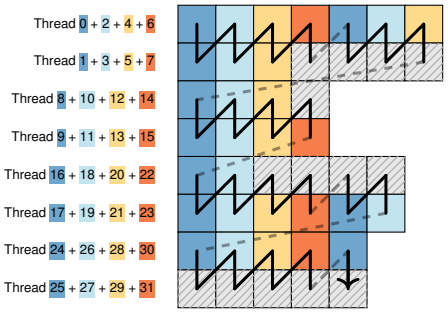
[6]The investigated GPUs have a warp size of 32. There are AMD GPUs with warp sizes of 64, but these are not discussed here.

[7]Maximum number of threads per block varies between GPUs

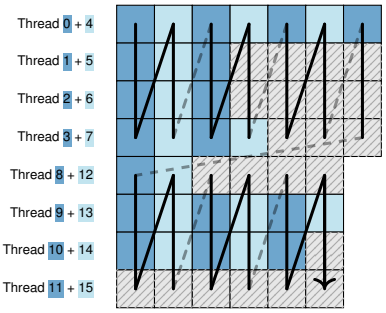
that such extreme matrices do not occur. Figure 3.5a and Figure 3.5b visualize how the shuffle SELL-$C$-$\sigma$ kernel operates on a sample matrix.

If the memory layout of SELL-$C$-$\sigma$ is supposed to remain unaltered, an inherent requirement of $N_{\text{threads\_per\_warp}} = C \cdot N_{\text{threads\_per\_row}}$ results, abbreviated as $N_{\text{tpw}}$ and $N_{\text{tpr}}$. In case this relation is violated, memory will either not coalesce correctly (see Figure 3.5d) or the *shfl* instructions will have to exchange data outside of their warp (see Figure 3.5c). We performed some attempts besides shared memory to circumvent this restriction by altering the memory layout to a row-major and column-major hybrid. However it ultimately ended up in causing edge cases which affected coalescing and accordingly poor performance. It should be noted that the dependence of $C$ on $N_{\text{tpr}}$ is not a problematic restriction. After all, SELL-$C$-$\sigma$ parameter $C$ should be chosen according to the compute architecture. There is no general reason why choosing an arbitrarily $C$ should increase performance. Moreover, it may even deteriorate performance, as more padding and matrix storage overhead is required.

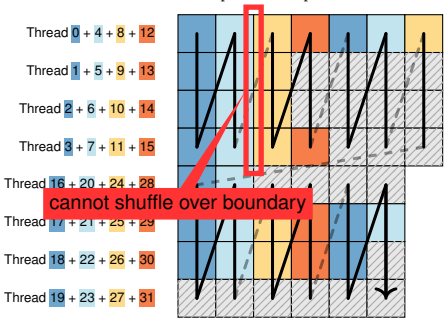
The final kernel with multiple threads per row works similarly to the naive approach, but stride and *shfl* reduction (see Figure 3.6) are added. Related performance results are plotted in Figure 3.7. On the first impression, these seem to be similar to the results observed in the previous section. Notable difference are as follows. First, RACE performance for 7pt on the L40 is ∼15% lower than with the single thread per row SELL-$C$-$\sigma$ implementation. Second, the RX 6900 performance actually increases for higher powers on 27pt.



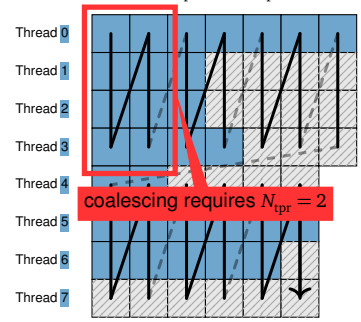

**(a)** SELL-2-1: $N_{\text{tpr}} = 4$, $N_{\text{tpw}} = 8$.




**(b)** SELL-4-1: $N_{\text{tpr}} = 2$, $N_{\text{tpw}} = 8$.




**(c)** SELL-4-1: $N_{\text{tpr}} = 4$, $N_{\text{tpw}} = 8$: Impossible reduction over warp boundary.




**(d)** SELL-4-1: $tpr = 1$, $N_{\text{tpw}} = 8$: Incorrect data layout.

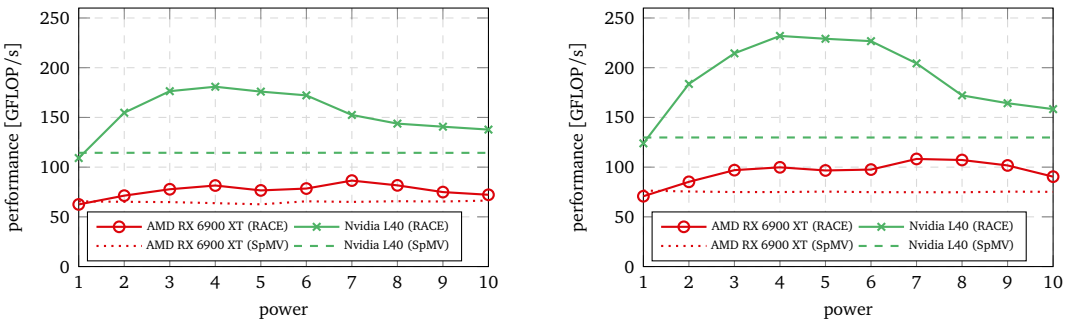
**Figure 3.5** – SELL-$C$-$\sigma$ on GPU example: Multiple threads per row are used. Dashed line mark warp boundary. (a) and (b) show valid parameters for $N_{\text{tpw}} = C \cdot N_{\text{tpr}}$. (c) and (d) are both invalid. (c) cannot work because the row result needs communication between two warps. (d) does not deliver good performance because data layout does not allow memory coaslescing.

```
1  void gpu_scs_shfl(val, col, chunkPtr, chunkLen, b, x) {
2      thread          = blockIdx.x * blockDim.x + threadIdx.x;
3      threadsPerChunk = THREADS_PER_ROW * C;
4      threadInChunk   = threadIdx.x % threadsPerChunk;
5      chunk           = thread / threadsPerChunk;
6      row             = threadInChunk % C + chunk * C;
7      idx             = row % C;
8      threadInRow     = threadInChunk / C;
9      ptr             = chunkPtr[chunk];
10     rowlen          = chunkLen[chunk];
11     sum             = 0.0;
12
13     for (i = threadInRow; i < rowlen; i += THREADS_PER_ROW) {
14         valColIdx = ptr + i * C + idx;
15         sum += val[valColIdx] * x[col[valColIdx]];
16     }
17
18     for (i = THREADS_PER_ROW >> 1; i > 0; i >>= 1)
19         sum += __shfl_down(sum, i * C);
20
21     if (threadInRow == 0)
22         b[row] = sum;
23 }
```

**Figure 3.6** – SELL-$C$-$\sigma$ shuffle kernel (*gpu-scs-cm-shfl*).

A decrease of 7pt stencil performance may be caused by scheduling overhead, as in this case there are four times as many threads to spawn, but that in itself should not cause problems. It is more likely that for a $N_{\text{tpr}} = 4$ and a $N_{\text{nzr}} = 7$ one thread may always be idling for the last three coefficients in a row. This could be a problem if all threads are saturated, since the GPU has to wait for these idling threads to finish before scheduling new threads. However since SpMV performance remains unchanged, we do not think that this causes the performance drop.

The RX 6900 experiences a similar drop, but only for powers near five. In addition, there is notable increase in performance with RACE for 27pt, but only for powers greater than five. We attribute this increase to better thread saturation, as higher powers will form smaller level groups, and thus smaller number of threads per kernel call. For example, on the RX 6900 a single threaded kernel at $p = 7$ will only launch ~34000 threads, which would not nearly saturate the available 81920. With $C = 8$ this count quadruples and easily saturates all threads.




**(a)** 3D 7pt-stencil (256x256x256). **(b)** 3D 27pt-stencil (128x128x256).

**Figure 3.7** – MPK-HIP: *gpu-scs-cm-shfl* ($C = 8, \sigma = 1$).

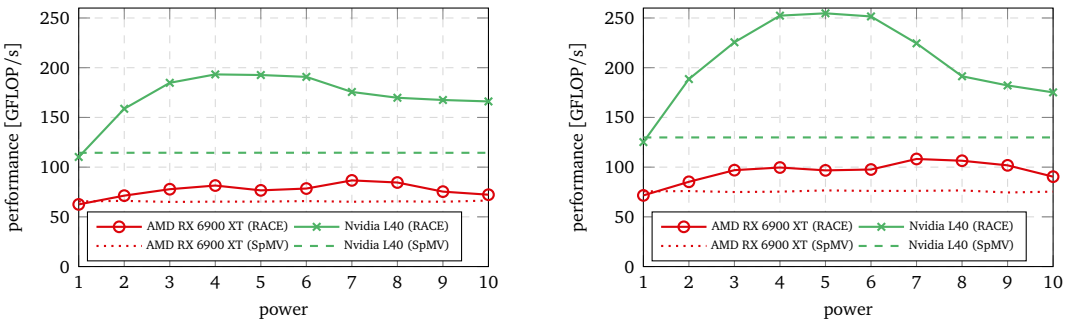
Concluding for this optimization step, using more threads results in small benefits, as long as the row length sufficiently large. In addition, more threads per row tend to get better performance if the thread count is too low per kernel call. We present more results with respect to thread count in Section 4.4.2.4, but deciding on a $N_{\text{tpr}}$ will ultimately depend on the matrix being used or power calculated.

### 3.2.8 HIP Graph

The last optimization step the SELL-$C$-$\sigma$ kernel is reducing call and synchronization overhead. Unfortunately it is not possible to avoid overhead entirely, but HIP (or CUDA) still can reduce overhead by the use of its Graph API. One of its main benefits is the support of fine grained workloads where each kernel call can depend on other kernel calls which forms a graph. This is a superset of the linear dependency chain that RACE requires. Thus the Graph API can still be used to replace many kernels calls with a single one, to ideally reduce call overhead. Generating this graph has an additional upfront cost, but it can be neglected when the same MPK is executed many times. To implement graph support, changes to the kernels themselves are not required. Instead, the regular kernel calls are not executed directly but recorded. Once this recording is completed, the generated graph can be executed by just a single call and it is possible to repeat it arbitrarily.

We implemented graph support in MPK-HIP and observed the results shown in Figure 3.8. For SpMV there is no observable difference for both GPUs and matrices, but for RACE there is a notable 20 GFLOP/s increase for $p > 3$ (L40). It is expected that SpMV does not yield a significant difference. After all, the same kernel calls are issued with and without graph, and each of these calls still needs synchronization at the end. The only way the Graph API can optimize execution is to perform ahead-of-time scheduling.

For SpMV the generated graph is likely too simple to yield a meaningful difference on any GPU. This is different to RACE, where a lot more kernel calls have to be issued. As one can see it does make a difference on the L40, but unfortunately this is not the case for the RX 6900. This is likely due to HIP's prematurity. In its current state the Graph API may solely exist to be API compatible with CUDA. We were able to measure performance regressions when using graphs on ROCm version 5.6 and older.




**(a)** 3D 7pt-stencil (256x256x256). **(b)** 3D 27pt-stencil (128x128x256).

**Figure 3.8** – MPK-HIP: *gpu-scs-cm-shfl, hip-graph* ($C = 8, \sigma = 1$).

### 3.2.9 Summary

By applying all the optimization steps above to our implementation, we are able to perform an MPK on a large matrix with significant speedup compared to an SpMV based MPK. Starting off the CRS implementation, the performance increases on the RX 6900 from 15 GFLOP/s to 110 GFLOP/s (best case) and on the L40 from 60 GFLOP/s to over 250 GFLOP/s (27pt stencil). Compared to the fully optimized SELL-$C$-$\sigma$ SpMV, the performance increase by using RACE is not as large, but for the L40 it is still a significant increase from 130 GFLOP/s to 255 GFLOP/s. On the RX 6900 it is only an increase from 75 GFLOP/s to 105 GFLOP/s. We discuss the reasons for these exact speedups in more detail in Section 4.4.3. In addition to that we will also compare the results with a larger variety of matrices, and the A100 and GH200 GPUs.

During implementation we also tested code optimization techniques such as unrolling and templating. However, we were not able to observe any meaningful differences with them. After all, the code is heavily bandwidth bound and spends most of its time waiting for memory loads. It would be rather surprising if reducing any other overhead would cause significant performance differences.

## 3.3 Benchmarking Infrastructure

Obtaining consistent benchmark results is crucial for drawing conclusions from hardware and software behavior. Since we had to run multiple benchmarks many times repeatedly with different parameters on different machines, we needed means to achieve that. We developed two tools, *bench.py* and *eval.py*. The purpose of *bench.py* being to submit benchmark jobs with configurable parameter combinations to multiple machines. Afterwards, *eval.py* would then processes their results and filters them based on a subset of the inspected parameters to generate graphs.

We separated these functionalities since malfunctions in a single tool would require the benchmark suite to rerun. These runs would sometimes take multiple hours to complete. In addition it enabled more flexible analysis by being able to filter parameters and to quickly recreate graphs. An overview of this process can be seen in Figure 3.9.

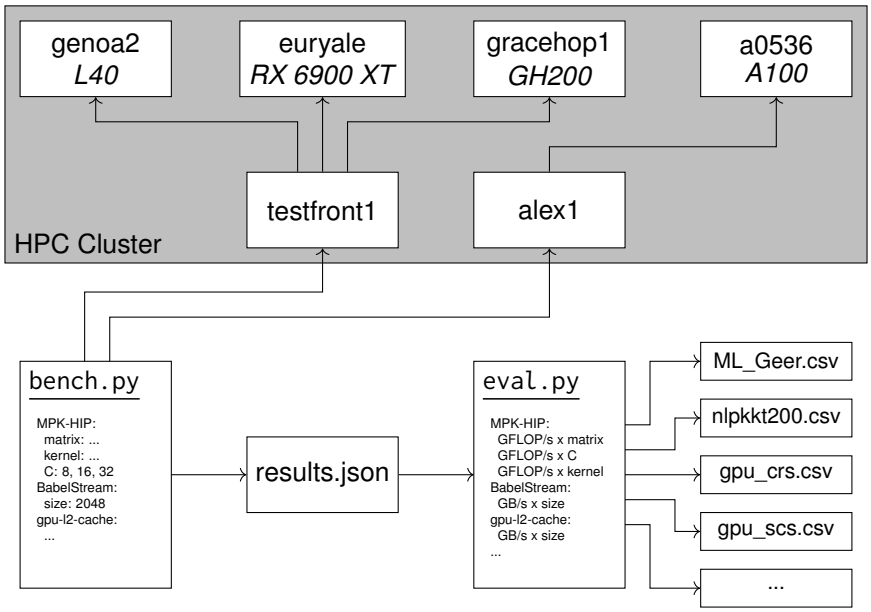



**Figure 3.9** – Benchmarking Infrastructure: *bench.py* connects to the cluster front ends, launches jobs, and collects the results. The results are then processed by *eval.py* to create plots.

# 4 ANALYSIS

## 4.1 Overview and Goals

In the previous chapter we discussed some promising performance improvements with MPK-HIP. However we have not yet provided a model that accurately predicts the measured performance figures. For example we noticed an approximate 2× best case increase on the L40 and a less but still substantial 1.4× increase on the RX 6900.

In order to outline the measured MPK and SpMV performance, we compare results of multiple independent benchmarks. The benchmarks we discuss are *BabelStream* for memory bandwidth measurement, *gpu-l2-cache* for cache bandwidth measurement, *MPK-HIP* for the main results, and *gpu-small-kernels* for ramp up analysis. In addition we assess the accuracy of the simplified SpMV roofline model from Section 2.4. We then show a refined version of this roofline model, and compare it against measurement results. We then proceed by attempting the same for MPK performance with RACE, followed by a discussion of encountered deviations. In the end we finish by doing a final comparison between all GPUs and their performance.

One main difference compared to the previous chapter is the introduction of new GPUs to our tests. We briefly hinted that performance on A100 and GH200 datacenter GPUs would not be good. To further explore the problems of MPKs we included them in our results. Note that we used the 80 GiB SXM4 model for the A100 and we used the 96 GiB GH200 “Grace Hopper” model for the H100.

## 4.2 BabelStream

Using purely theoretical DRAM bandwidth numbers (see Section 2.4) may provide a good approximation, but it is possibly misleading for the real world performance. Hardware overhead is usually ignored, and programmatically achievable bandwidth may not necessarily match the nominal figures. The *BabelStream* benchmark is a commonly used benchmark to obtain real-world memory performance on GPU, previously called the *GPU-STREAM* [5] benchmark. It is not limited to get the maximum sustained bandwidth. It is also usable to determine the ramp up behavior, which is an important factor for kernel performance with few threads. A GPU can have an arbitrarily high sustained bandwidth and may still be unusable, if its beneficial use is limited to unrealistically large problem sizes.

As can be seen in Table 4.1 the real world performance may be 20% less than nominal in some cases (see Table 2.1). For example, the RX 6900 might get only ∼80% of the nominal performance for *Add* and *Triad*, being the same for the GH200 on *Copy* and *Mul*. Due to different utilization

| GPU | GB/s (Copy) | GB/s (Mul) | GB/s (Add) | GB/s (Triad) | GB/s (Dot) |
|---|---|---|---|---|---|
| AMD RX 6900 XT | 458 | 456 | 415 | 415 | 488 |
| Nvidia L40 | 775 | 769 | 768 | 802 | 838 |
| Nvidia A100 (80 GiB) | 1746 | 1746 | 1771 | 1773 | 1752 |
| Nvidia GH200 (96 GiB) | 3215 | 3207 | 3519 | 3523 | 3744 |

**Table 4.1** – BabelStream memory performance (-s $1024 \cdot 2^{19}$).

and data dependencies it is possible that the hardware is not able to hide memory latency. While of course the exact memory performance will still differ for different GPU kernels, the worst case figures in Table 4.1 may still be a good indicator for a roofline analysis.

One more important metric to test against is ramp up behavior, which means how fast the GPU reaches its sustained memory performance when increasing the number of elements. Because BabelStream uses exactly one thread per element, this also gives insight about how quickly the GPU is saturated with threads. More elements per thread might scale differently, but BabelStream only provides kernels with one element per thread.

Figure 4.1 demonstrates how the *Copy* stream scales on different GPUs. On the left all GPUs perform at lowest bandwidth, since the problem size is too small to saturate the GPU. On the right all GPUs reach the maximum bandwidth limited by DRAM. The latter words are rather carefully chosen, as the maximum DRAM bandwidth does, as shown in the graph, not truly represent the maximum achievable bandwidth. Both the RX 6900 and L40 show a peak in the middle which is above the maximum sustained bandwidth for larger data. This is because both GPUs ramp up fast enough for small enough data to provoke cache bandwidth instead of DRAM bandwidth. Of course these peaks do not show the real maximum cache bandwidth, as a stream-like benchmark is not suitable to correctly determine it. Cache performance will be discussed in more detail in Section 4.3. As expected, the graphs converge to identical values as shown previously in Table 4.1.

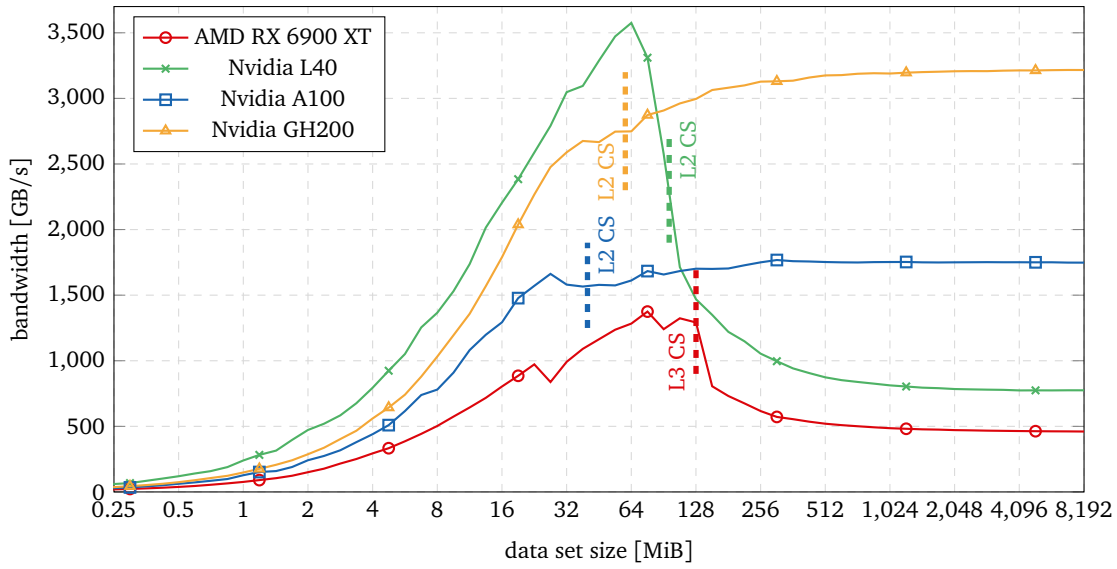



**Figure 4.1** – BabelStream bandwidth (double precision, copy): Dashed lines show LLC size.

Another important effect is the ramp up speed. One can clearly see the L40 being the fastest with the smallest data set, even though GPUs like the A100 and GH200 are considered to be GPUs with higher performance. This is likely because the L40 uses a consumer chip, which is built primarily for computer graphics. Thus faster ramp up may be more important for the L40 compared to the A100 and GH200, which are specifically built for datacenter applications. Out of all GPUs the RX 6900 is not only the slowest, but it also ramps up slower than all others. That is even though it also is a graphics oriented GPU and has a larger cache than the L40.

Having good ramp up behavior is crucial for running small kernels at good performance. We already briefly discussed thread saturation as factor in Section 3.2, but in addition there is also base cost for each kernel call. As we will discuss later in Section 4.4.2.4, this poses a limitation for how well the RACE MPK is able to operate on GPU.

Last, the performance drops occur as expected once the data set size exceeds the cache size. The RX 6900 has 128 MiB of cache, which is exactly where the graph starts to substantially drop off. For the L40 this also occurs approximately at its cache size of 96 MiB. We expected to see a peak for the A100 at 50 MiB and GH200 at 60 MiB, but no such peak is visible. Both GPUs already run into the DRAM bandwidth limit before exceeding their cache size. The A100 has a tiny local peak at 28 MiB, but it is too small to be of significance. This behavior on the A100 and GH200 further hints why we did not run our initial optimization benchmarks on the A100 and GH200.

## 4.3 gpu-l2-cache

As shown in the previous section, measuring memory bandwidth may be influenced by cache behavior. If more detailed information on cache sizes and cache speed is required, a more precise and targeted benchmark compared to BabelStream is needed. RACE MPKs work by reusing data from caches to avoid loading them from memory multiple times. Expecting these caches access to be negligibly fast would be naive to assume. Consequently we have to precisely measure how fast the caches are to reasonably predict the MPK performance (see Section 4.4.3). For this, and to find out the data boundaries at which caches incur a performance penalty, we use the *gpu-l2-cache* benchmark [9]. Although its name might suggest otherwise, it is also able to measure cache performance of the L3 cache present on the RX 6900, not just L2.

*gpu-l2-cache* determines the bandwidth for a given data set size by doing the following. A kernel call is issued with a large number of threads to ensure the GPU is saturated. Each thread then loads 32 double precision values from a buffer which is sized accordingly to the data set to test. The data is read in a spread pattern to avoid caching such that the data is unlikely to be read from cache, if the overall data set does not fit in a cache level.

Figure 4.2 shows the bandwidth observed as by *gpu-l2-cache* over an increasing data set size. As expected all GPUs show declining bandwidth as data set size is increased. All values converge against their DRAM bandwidth limit for large data size. The exact bandwidth observed (see Table 4.2) is actually slightly higher than the maximum of BabelStream. Compared to operations performed by BabelStream, *gpu-l2-cache* is most comparable to *Dot*, as this is the only operation which does not write to global memory. The only real difference is that *gpu-l2-cache* does not even write to shared memory and that it performs more reads per thread than *Dot*.

Cache boundaries are clearly distinguishable in **??**. We recall that the RX 6900 has 128 MiB of L3 cache. This is exactly where the bandwidth drop is visible. The same applies for the L40, which drops at 96 MiB. In addition to the L3 to DRAM drop on the RX 6900, there is another drop for the transition from L2 to L3 at $\sim$6 MiB. This is slightly higher than the real L2 cache size of 4 MiB.

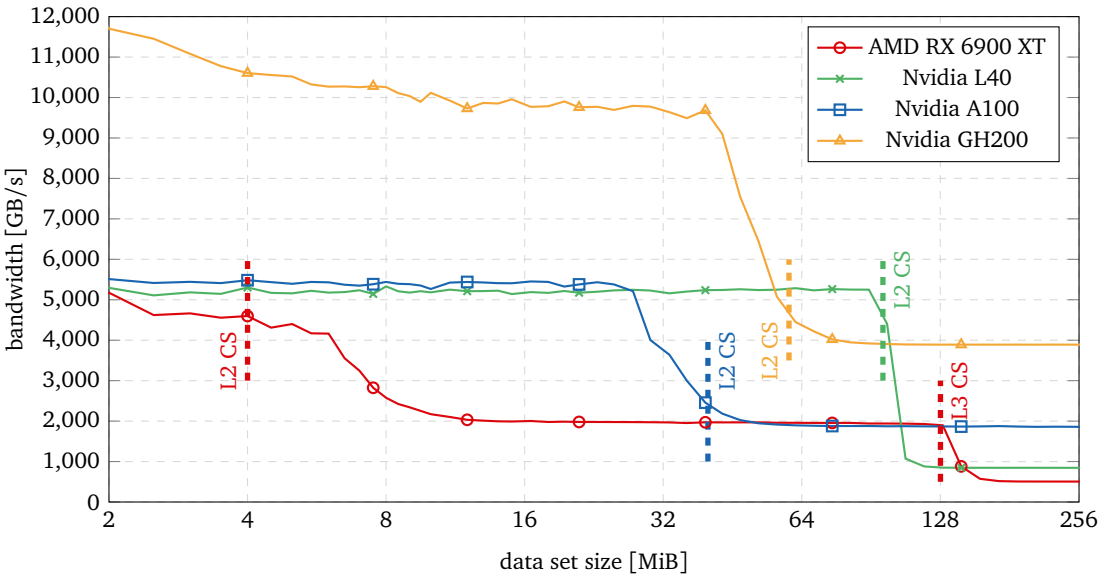



**Figure 4.2** – gpu-l2-cache bandwidth: Dashed lines mark the real cache size.

| GPU | GB/s (LLC) | GB/s (DRAM) |
|---|---|---|
| AMD RX 6900 XT | 1950 | 491 |
| Nvidia L40 | 5286 | 833 |
| Nvidia A100 (80 GiB) | 5325 | 1778 |
| Nvidia GH200 (96 GiB) | 9715 | 3883 |

**Table 4.2** – *gpu-l2-cache* DRAM and LLC bandwidth at respective plateaus. LLC refers to L2 on Nvidia and L3 on AMD.

However, the L2 to L3 drop is not important for the scope MPKs, since it will be too small for being relevant for cache blocking.

Both the A100 and GH200 show a significant drop as well, but it occurs earlier as their caches are not as big. Instead, they already experience a drop before their real cache size. This earlier than expected drop may be due to the two-way cache partitioning the A100 is using, and likely the GH200 experiences a similar effect. Another possibility is a more aggressive eviction strategy, since the drop slope ends precisely at the cache's real size.

The most significant observation in the graphs are the speedups. The large caches of the RX 6900 and L40 are clearly measurable, and bandwidth becomes about six times as fast for data that resides in cache. Albeit still present, the speedup on the A100 and GH200 from going to DRAM to cache is not as large and is "only" two to three times. One more thing that should be mentioned with respect to the L40 cache bandwidth is that the bandwidth from the L2 cache would saturate at ~4 TB/s due to a compute limit for earlier revisions of the benchmark. Although both L40 and RX 6900 use consumer chips, this problem only occurred on the L40, as its double precision compute to bandwidth ratio is slightly lower compared to all other GPUs.

For the RX 6900[8] and L40 we observed a fast ramp up and significant speed ups compared to DRAM performance. This already shows why the cache blocking with RACE can perform well. The cache bandwidth speedup on the RX 6900 is slightly less than on the L40, so one might assume that the RX 6900 should experience similar speedup when using RACE. As we already briefly discussed the performance of RACE in Section 3.2.8, the performance gain is in fact not as high. While there is measurable speedup, it is much lower. We discuss the exact influence of the cache bandwidth in Section 4.4.3 in detail. Here it is noted, that the practically usable cache bandwidth on the RX 6900 is not as high as *gpu-l2-cache* measures it.

## 4.4 MPK-HIP

### 4.4.1 SpMV Performance

#### 4.4.1.1 Performance Model

Previously, we created an approximate roofline model for a classic SpMV kernel which assumed a data intensity of 6 bytes per FLOP. This is a reasonable approximation, but not accurate enough for a detailed analysis. For example, the previous model assumes x to be always cached, which is not true in practice.

In the context of our new model, we use a different definition for *cached* and *uncached*. Unlike normal bandwidth analysis, we have to carefully differentiate between DRAM and LLC bandwidth. So it must be reasoned well whether a memory operation is cached at all, and if it is cached, from which level cache it will be fetched from. In the following paragraph we will refer to *cached*, if data can be fetched from L1, with the exception being the RX 6900, as its L2 cache is much smaller than all Nvidia GPUs[9]. We will use the term *uncached*, if something is either fetched from DRAM or LLC. Whether something will be fetched from LLC or DRAM will then depend on RACE's execution order.

Looking at the kernel more closely (see previous Figure 3.6), there are six buffers being accessed in total: *val*, *x*, *col*, *b*, *chunkPtr*, and *chunkLen*. The latter two are being accessed $N_{\text{tpw}}$ times (for best speed) per chunk, but we will assume that all but the first read will be *cached*. *b* is only accessed once per row, when the result is written back to memory, and thus is *uncached*. As each matrix coefficient is only processed once, *val* and *col* are always *uncached*. For a diagonal matrix, which RACE attempts to transform to the input matrix, we assume that subsequent rows always reuse all but one value from *x*. Consequently only a single access to *x* is *uncached* per row. With that knowledge we can derive the following formulas:

$$W_r = 2N_{\text{nzr}}\text{FLOP} \tag{4.1}$$

$$Q_r = \frac{Q_{\text{chunkPtr}} + Q_{\text{chunkLen}}}{C} + Q_{b_r} \quad + Q_{\text{val}_r} \quad + Q_{\text{col}_r} \quad + Q_{x_r} \tag{4.2}$$

$$Q_r = \frac{4\,\text{Byte} + 4\,\text{Byte}}{C} \quad + 8\,\text{Byte} + N_{\text{nzr}} \cdot 8\,\text{Byte} + N_{\text{nzr}} \cdot 4\,\text{Byte} + 8\,\text{Byte} \tag{4.3}$$

$$Q_r \quad = \frac{8\,\text{Byte}}{C} \quad + N_{\text{nzr}} \cdot 12\,\text{Byte} + 16\,\text{Byte} \tag{4.4}$$

$$D = \frac{Q_r}{W_r} = \frac{4\,\text{Byte}}{CN_{\text{nzr}}\text{FLOP}} + 6\,\frac{\text{Byte}}{\text{FLOP}} \quad + \frac{8\,\text{Byte}}{N_{\text{nzr}}\text{FLOP}} \tag{4.5}$$

[8] Although the RX 6900 has the slowest ramp up, it is still able to ramp up to use the cache.

[9] In this paragraph we refer to caching between subsequent rows for classic SpMV, not level groups from RACE

#### 4.4.1.2 Performance Measurements

Any performance model is only useful, if it matches with measured results. In contrast to the implementation phase in Section 3.2, we now consider a wide range of matrices with different characteristics. For bandwidth we assume the measured DRAM bandwidth from *gpu-l2-cache* (see Table 2.3). As a good reference we will include a custom matrix, which we call *band-XY*. This matrix is perfect band matrix and attempts to eliminate any other factors which may be cause by the matrix shape. Our model should cover this case very accurately. In addition we, we use the previous 7pt and 27pt stencil matrices and a few other selected ones from the SuiteSparse Matrix Collection [31]. For bandwidth, we only assume DRAM bandwidth, as in this step we only measure a classic SpMV MPK and all matrices are large enough to not fit in the cache. While we do benchmark a classic SpMV, we do so with RACE's permutation already applied, which should make our model more accurate.

Table 4.3 shows the properties of the matrices under test, including $D_{C=8}$ which is needed to resolve (4.5). The measurement results compared with the estimations can be seen in Figure 4.3. Black ticks denote the calculated estimates using the previous formula, while the bars show the actual measurement results. The RX 6900 and the L40's measurements match the model pretty closely for most matrices. On the other hand, the A100 and GH200 are roughly off by 10%-30%.

The results can be categorized in two groups. Matrices which run below expected performance for all GPUs, and those, where only A100 and GH200 perform below expectation. The most extreme matrices are *delaunay_n24* and *kkt_power*, where all GPUs perform poorly. Both matrices are extreme in the sense, that they have a low number of rows which and an unusually high $N_{\text{nzr}}$. Since our model is based on an ideal diagonal matrix, which has a constant $N_{\text{nzr}}$, suboptimal performance is expected. *cage14* is an outlier, as it also has a large variation in $N_{\text{nzr}}$, but since it is distributed more evenly, the results are not severely affected.

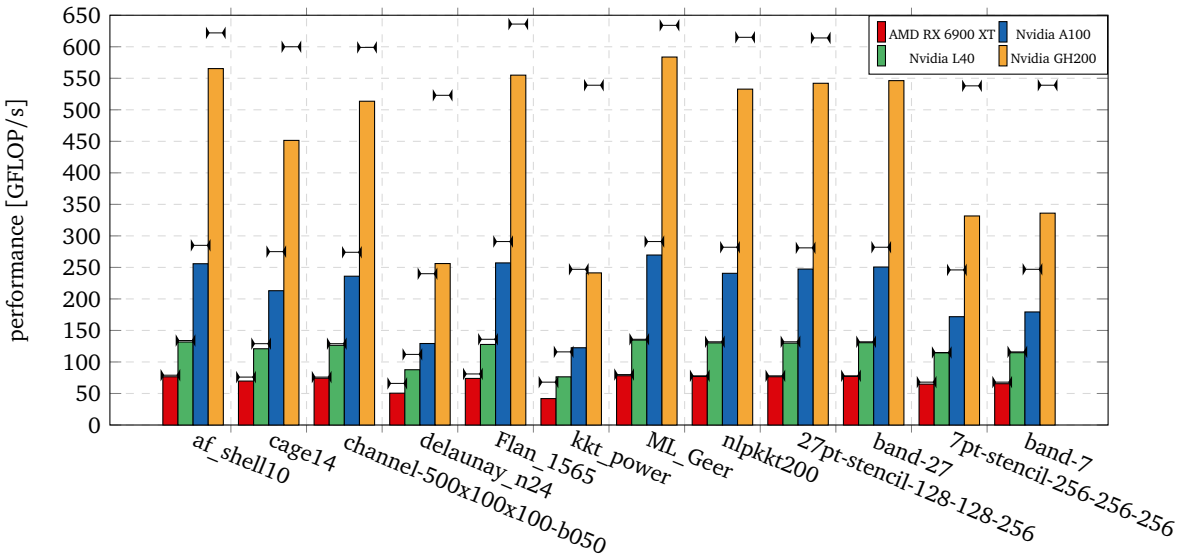



**Figure 4.3** – MPK-HIP: classic SpMV, misc matrices, $C = 8$, *gpu-scs-cm-shfl*, $p = 5$, black ticks show estimated numbers by Equation (4.5).

| Matrix | $N_r$ | $N_{nz}$ | $\overline{N_{nzr}}$ | $D_{C=8}$ | SELL-8-1 size |
|---|---|---|---|---|---|
| af_shell10 | 1508065 | 52672325 | 34.93 | 6.24 | 634.6 MB |
| audikw_1 | 943695 | 77651847 | 82.28 | 6.10 | 1092.3 MB |
| bmw7st_1 | 141347 | 7339667 | 51.93 | 6.16 | 96.1 MB |
| bone010 | 986703 | 71666325 | 72.63 | 6.12 | 919.3 MB |
| bundle_adj | 513351 | 20208051 | 39.36 | 6.22 | 320.9 MB |
| cage14 | 1505785 | 27130349 | 18.02 | 6.47 | 343.9 MB |
| cant | 62451 | 4007383 | 64.17 | 6.13 | 52.8 MB |
| cfd2 | 123440 | 3087898 | 25.02 | 6.34 | 39.6 MB |
| channel-500x100x100-b050 | 4802000 | 85362744 | 17.78 | 6.48 | 1036.3 MB |
| crankseg_1 | 52804 | 10614210 | 201.01 | 6.04 | 145.1 MB |
| delaunay_n24 | 16777216 | 100663202 | 6.00 | 7.42 | 1628.7 MB |
| dielFilterV3real | 1102824 | 89306020 | 80.98 | 6.10 | 1596.7 MB |
| Emilia_923 | 923136 | 41005206 | 44.42 | 6.19 | 515.5 MB |
| F1 | 343791 | 26837113 | 78.06 | 6.11 | 382.9 MB |
| Fault_639 | 638802 | 28614564 | 44.79 | 6.19 | 365.2 MB |
| Flan_1565 | 1564794 | 117406044 | 75.03 | 6.11 | 1492.7 MB |
| G3_circuit | 1585478 | 7660826 | 4.83 | 7.76 | 97.2 MB |
| gearbox | 153746 | 9080404 | 59.06 | 6.14 | 127.0 MB |
| Geo_1438 | 1437960 | 63156690 | 43.92 | 6.19 | 799.5 MB |
| Graphene-8192-8192 | 67108864 | 872235016 | 13.00 | 6.65 | 10535.8 MB |
| gsm_106857 | 589446 | 21758924 | 36.91 | 6.23 | 264.1 MB |
| Hamrle3 | 1447360 | 5514242 | 3.81 | 8.23 | 67.7 MB |
| hollywood-2009 | 1139905 | 113891327 | 99.91 | 6.09 | 2333.4 MB |
| Hook_1498 | 1498023 | 60917445 | 40.67 | 6.21 | 973.1 MB |
| HPCG-128-128-128 | 2097152 | 55742968 | 26.58 | 6.32 | 673.8 MB |
| inline_1 | 503712 | 36816342 | 73.09 | 6.12 | 527.8 MB |
| kkt_power | 2063494 | 14612663 | 7.08 | 7.20 | 278.3 MB |
| ldoor | 952203 | 46522475 | 48.86 | 6.17 | 626.2 MB |
| Lynx649 | 64950632 | 978866282 | 15.07 | 6.56 | 12961.6 MB |
| ML_Geer | 1504002 | 110879972 | 73.72 | 6.12 | 1335.7 MB |
| nlpkkt200 | 16240000 | 448225632 | 27.60 | 6.31 | 5419.0 MB |
| offshore | 259789 | 4242673 | 16.33 | 6.52 | 63.5 MB |
| parabolic_fem | 525825 | 3674625 | 6.99 | 7.22 | 44.7 MB |
| pwtk | 217918 | 11634424 | 53.39 | 6.16 | 141.4 MB |
| rajat31 | 4690002 | 20316253 | 4.33 | 7.96 | 278.7 MB |
| RM07R | 381689 | 37464962 | 98.16 | 6.09 | 502.3 MB |
| road_usa | 23947347 | 57708624 | 2.41 | 9.53 | 1028.1 MB |
| Serena | 1391349 | 64531701 | 46.38 | 6.18 | 886.2 MB |
| ship_003 | 121728 | 8086034 | 66.43 | 6.13 | 107.1 MB |
| stokes | 11449533 | 349321980 | 30.51 | 6.28 | 5408.8 MB |
| thermal2 | 1228045 | 8580313 | 6.99 | 7.22 | 120.5 MB |
| Topi-real-256-256-256 | 67108864 | 802160640 | 11.95 | 6.71 | 9714.9 MB |
| webbase-1M | 1000005 | 3105536 | 3.11 | 8.74 | 63.7 MB |
| xenon2 | 157464 | 3866688 | 24.56 | 6.35 | 50.6 MB |
| 7pt-stencil-256-256-256 | 16777216 | 117047296 | 6.98 | 7.22 | 1424.5 MB |
| band-7 | 16777216 | 117440500 | 7.00 | 7.21 | 1426.0 MB |
| 27pt-stencil-128-128-256 | 4194304 | 111777784 | 26.65 | 6.32 | 1351.8 MB |
| band-27 | 4194304 | 113246026 | 27.00 | 6.31 | 1363.1 MB |

**Table 4.3** – Matrix properties: $N_r$ is number of rows, $N_{nz}$ is number of non zeroes, size is the number of megabytes for a SELL-8-1 encoded matrix, $\overline{N_{nzr}}$ is the average number of non-zeroes per row, $D_{C=8}$ is the calculated average data intensity in Bytes per FLOP for $C = 8$.

For the matrices performing as expected on the RX 6900 and L40 but not on A100 and GH200, the performance can be improved by adjusting $C$. When choosing $C = 32$ for *band-7*, the performance is improved to within 85% of the A100's and GH200's expected values.

To conclude, all matrices generally perform as predicted in (4.5), if the matrices do not have extreme shapes. In these cases the DRAM bandwidth measured previously (see Table 4.2) matches the results very well. This is surprising to a certain extent, as even BabelStream is not able to achieve maximum bandwidth and BabelStream's kernels are already as basic as possible.

#### 4.4.1.3 In-Cache Performance

In the previous section we demonstrated that the maximum achievable DRAM performance is in accordance with *Pgpu-l2-cache*'s measurement results for the RX 6900 and L40. In order to extend the refined SpMV model to RACE, the LLC bandwidth needs to be determined. Modelling the performance with RACE is rather complex, as it involves mixed DRAM and LLC bandwidth. Both DRAM and LLC figures were already measured via *gpu-l2-cache*, but as observed with *BabelStream* the real world bandwidth may differ for other kernels. To avoid uncertainty of pure in-cache performance with regard to the SpMV kernel, we perform the same benchmark as in the previous chapter. However, previous matrices were explicitly chosen to not fit in the cache, so in this measurement, the opposite needs to be the case. The matrices now have to be carefully chosen in order to be large enough, but not too large to exceed LLC size. This precise size restriction limits us to just synthetic matrices. We use a *band-27* matrix as before, but now $N_r$ is varied in order to achieve the desired matrix size. To avoid performance drops earlier than expected and to further analyze ramp up effects, we plot performance over an increasing matrix size instead of using a single fixed size.

When looking at Figure 4.4, multiple effects are visible. The ramp up is similar to values measured with BabelStream. As more and more threads start to process the matrix, performance increases. Once performance has peaked, the RX 6900 and L40 show a comparable drop as for BabelStream. The modelled in-cache peak performance with *gpu-l2-cache*'s measured bandwidth is marked by

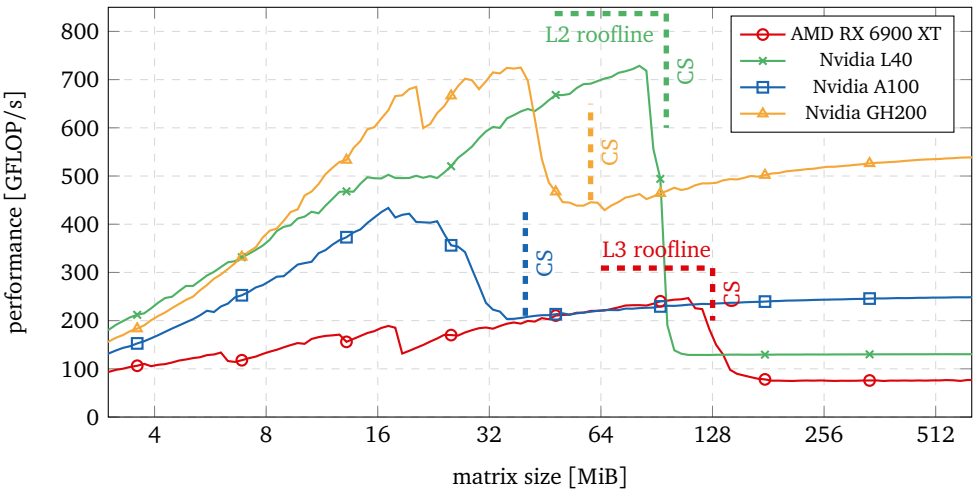



**Figure 4.4** – MPK-HIP: classic SpMV, band-27, $C = 8$, *gpu-scs-cm-shfl*, $p = 5$. Dashed lines show cache size and in-cache roofline, A100 and GH200 rooflines are not shown due being outside the scale.

dashed lines. The RX 6900 does not reach the expected 309 GFLOP/s, which is ∼60 GFLOP/s too low. Similarly the L40 only reaches 728 GFLOP/s instead of the estimated 837 GFLOP/s.

The behavior of the A100 and GH200 is more interesting. No expected peak performance is shown for these because the measured performance is severely below the expected 843 GFLOP/s (A100) and 1539 GFLOP/s (GH200). A significant ramp up is visible, as long as the matrix still fits into the cache of the GPU. With BabelStream the bandwidth nowhere exceeded the DRAM's maximum (see Figure 4.1). The naive assumption at this point would be, that the A100 and GH200 should benefit from cache blocking since the SpMV kernel is able to run faster when only using data from cache. We discuss cache blocking performance with RACE more in detail in the following sections, but it should be noted that in contrast to the RX 6900 and the L40, the performance gain inside the cache compared to outside is much lower. On the RX 6900 the speedup is ∼3.3× while on the L40 the speedup is as high as ∼5.6×. In comparison, the A100 does not even achieve a speedup of 2× and the GH200 not even reach 1.5×. Considering that the matrix size window with achievable speedup is much larger on the RX 6900 and the L40, actual speedup on cache blocked MPKs become even more unlikely.

## 4.4.2 RACE Performance

### 4.4.2.1 Performance Model

In the previous section we learned that the SpMV kernel can experience speedup when running entirely in-cache. The next step is to create a performance model with regard to cache blocking via RACE. As we have in Section 2.8, speedup via cache blocking with RACE is achieved by reusing data from cache for successive powers. Calculating the first power ($p = 1$) is always uncached[10], while the following power calculations use data from the cache only. While the model of (4.5) is still applicable in terms of data intensity, we have to differentiate two phases during calculating of $P = WT^{-1}$. We define a new $P$ by taking into account the ratio of uncached to cached work, which depends on the power to calculate:

$$P = \frac{W}{T} \quad , \quad \frac{1}{p} = \frac{W_{\text{dram}}}{W_{\text{dram}} + W_{\text{cache}}} \quad , \quad 1 - \frac{1}{p} = \frac{W_{\text{cache}}}{W_{\text{dram}} + W_{\text{cache}}} \tag{4.6}$$

$$P = \frac{W_{\text{dram}} + W_{\text{cache}}}{T_{\text{dram}} + T_{\text{cache}}} = \frac{1}{\frac{T_{\text{dram}}+T_{\text{cache}}}{W_{\text{dram}}+W_{\text{cache}}}} = \frac{1}{\frac{T_{\text{dram}}}{W_{\text{dram}}+W_{\text{cache}}} + \frac{T_{\text{cache}}}{W_{\text{dram}}+W_{\text{cache}}}} \tag{4.7}$$

$$= \frac{1}{\frac{\frac{W_{\text{dram}}}{W_{\text{dram}}+W_{\text{cache}}}}{\frac{W_{\text{dram}}}{T_{\text{dram}}}} + \frac{\frac{W_{\text{cache}}}{W_{\text{dram}}+W_{\text{cache}}}}{\frac{W_{\text{cache}}}{T_{\text{cache}}}}} = \frac{1}{\frac{\frac{1}{p}}{P_{\text{dram}}} + \frac{1-\frac{1}{p}}{P_{\text{cache}}}} \tag{4.8}$$

The final performance can then be calculated inserting the respective DRAM and LLC performance figures into (4.5). For $p = 5, C = 8$ and the L40 as an example, this results in the following performance for *band-27*:

[10] In this context we refer *cached* and *uncached* with respect to the LLC.

$$P_{\text{L40,dram}} = \frac{B_{\text{L40,dram}}}{D} = \frac{833\ \frac{\text{GByte}}{\text{s}}}{6.31\ \frac{\text{Byte}}{\text{FLOP}}} \approx 132.0\ \frac{\text{GFLOP}}{\text{s}} \tag{4.9}$$

$$P_{\text{L40,cache}} = \frac{B_{\text{L40,cache}}}{D} = \frac{5286\ \frac{\text{GByte}}{\text{s}}}{6.31\ \frac{\text{Byte}}{\text{FLOP}}} \approx 837.7\ \frac{\text{GFLOP}}{\text{s}} \tag{4.10}$$

$$P_{\text{L40}} = \frac{1}{\frac{\frac{1}{5}}{132.0\ \frac{\text{GFLOP}}{\text{s}}} + \frac{1-\frac{1}{5}}{837.7\ \frac{\text{GFLOP}}{\text{s}}}} \approx 404.8\ \frac{\text{GFLOP}}{\text{s}} \tag{4.11}$$

#### 4.4.2.2 Performance Measurements

We now compare the RACE model from the previous section to real measurements. Figure 4.5 shows the performance with RACE (bars) against the model (black ticks). As can be seen, the measured performance is a lot worse compared to what the previous model suggests. On the RX 6900, the performance deviates by approximately a relative error of ~2, while the L40 fits to the model best with a relative error of ~0.3 only. The relative error on the A100 and GH200 is even higher. Consequently it is not visible in the chart due to scale.

Some matrices perform worse than others, similar as they do with classic SpMV (see Section 4.4.1.2). However, the performance with RACE generally varies a lot more. Considering the overall error of this model raises the question, why it deviates so much. While it does correctly consider cache and uncached performance, it assumes that peak bandwidth, thus performance, is achieved at all times. Unfortunately this is a flawed assumption. In the next section a more realistic model is presented, which explains this behavior.

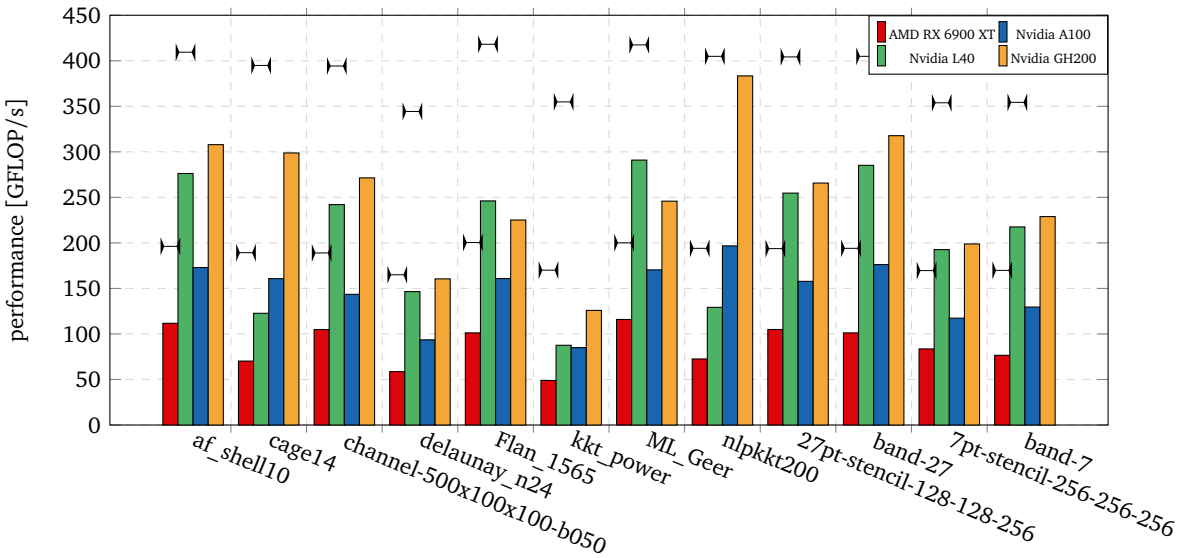



**Figure 4.5** – MPK-HIP: RACE, misc matrices, $C = 8$, *gpu-scs-cm-shfl*, $p = 5$, black ticks show estimated numbers by Equation (4.8).

#### 4.4.2.3 Measurement-Based Explanation

The naive performance model for RACE is nice, because it suggests big theoretical performance increases for all GPUs with fast cache. In practice the situation is different, because ramp up behavior has not yet been considered.

When looking back to Figure 2.7, calculating a matrix power via RACE involves many smaller execution steps. The exact number of execution steps is $p \cdot N_{\mathrm{LG}}$ compared to just $p$ for regular SpMV. Generally the larger the cache or the lower the power is, the fewer level groups ($N_{\mathrm{LG}}$) are formed. In opposite, if there are increasingly more execution steps, more GPU kernel calls have to be issued. However trading more kernel calls for better performance eventually hits into a ramp up limit, because the kernel calls are not able use the full bandwidth.

Figure 4.6 visualizes the ramp up behavior of RACE, while being overlaid with the previous SpMV ramp up (dotted). For smaller matrix sizes the performance with RACE is identical with SpMV, but for increasingly large matrices, while still remaining under the cache size, dips are repeatedly observed. These dips are caused by RACE increasing the number of level groups $N_{\mathrm{LG}}$, which in turn causes an increasing amount of kernel calls. For example, the first dip occurs at the point, where RACE increases $N_{\mathrm{LG}}$ to two, which happens earlier for GPUs with smaller cache. While it is not as accurate in practice, we assume that each kernel only processes half the matrix from that point. When there are now twice the number of kernel calls, with each SpMV only processing a matrix half the size, the performance drops approximately to the level measured for SpMV at half that size. The same occurs for all other dips at the integer multiple of the first drop. It should be noted, that for the RX 6900 this repeated dipping is overlaid with the same small performance drops already occurring for SpMV.

Unfortunately this means that for smaller cache sizes there simply are too many level groups, thus too many kernel calls for the performance to ramp up properly. Therefore it is not possible to properly model this behavior by peak bandwidth, as it is never reached. This severely affects the A100 and GH200, and causes their RACE performance for large matrices to be much worse than classic SpMV. Only on the L40, and to a small degree on the RX 6900, the performance

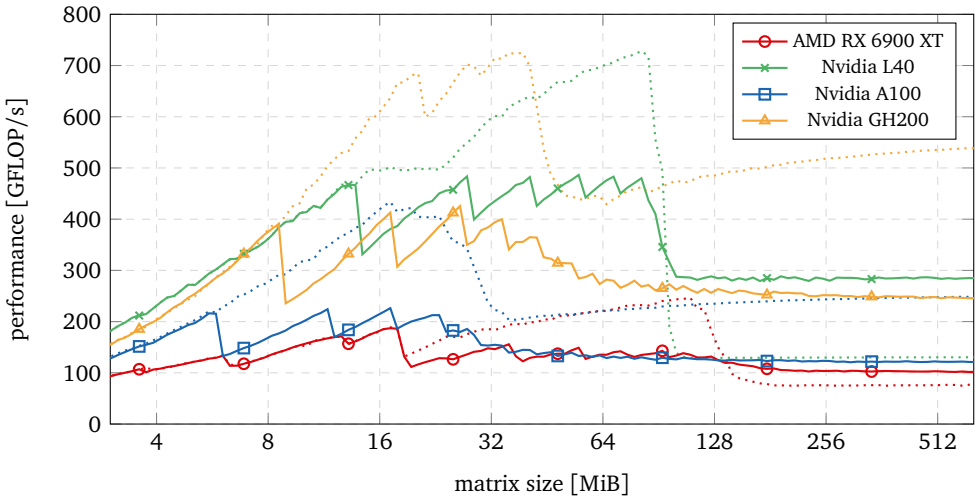


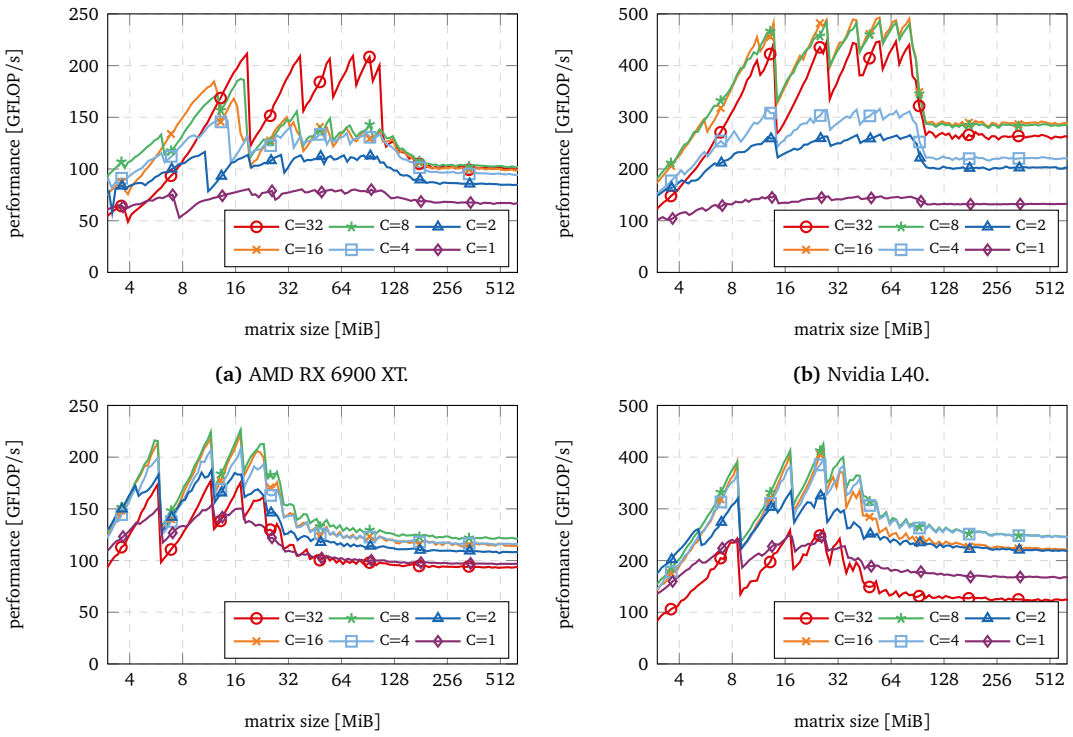

**Figure 4.6** – MPK-HIP: RACE, band-27, $C = 8$, *gpu-scs-cm-shfl*, $p = 5$. Dotted lines show previous classic SpMV results. The sawtooth-like dips show when RACE increases the number of level groups.

ramps up early enough to perform properly within the cached window. Using these peaks in the cache performance figures, these can be combined with the standard SpMV performance using (4.8). For $P_{\text{cache,L40}} = 480\,\text{GFLOP/s}$ and $P_{\text{cache,RX6900}} = 150\,\text{GFLOP/s}$ we get $P_{\text{L40}} \approx 314\,\text{GFLOP/s}$ and $P_{\text{RX6900}} \approx 125\,\text{GFLOP/s}$. Both values are still too high but much closer to real world performance. The accuracy could probably be further improved by choosing $P_{\text{cache}}$ more conservatively, but this couldbe fifficult, as for example the RX 6900 performance is really inconsistent.

#### 4.4.2.4 Ramp Up Reevaluation

In the previous section we concluded that performance cannot reach peak bandwidth under current circumstances, and thus performance. One possibility of coping with the early increase of $N_{\text{LG}}$ is to improve the ramp up. We briefly touched this in Section 3.2.7, where we tried to increase the number of threads per matrix row in order to improve GPU saturation. In theory this should work equally well for improving ramp up, but we only showed results for $C = 8$ and thus $N_{\text{tpr}} = 4$ in the previous graphs.

Figure 4.7 shows ramp up behavior for all possible numbers of threads per row $NN_{\text{tpr}}$. We recall that $N_{\text{tpw}} = C \cdot N_{\text{tpr}}$ and thus $N_{\text{tpr}}$ and $C$ are inverse proportional. An important takeaway from this measurement is that performance does not ramp up better for $C < 8$ (i.e. $N_{\text{tpr}} > 4$). It would be more appropriate to say the opposite is the case. However, we should remember that the higher $N_{\text{tpr}}$ is, the more overhead per row there is, as more threads will run idle.




**(a)** AMD RX 6900 XT.

**(b)** Nvidia L40.

**(c)** Nvidia A100.

**(d)** Nvidia GH200.

**Figure 4.7** – MPK-HIP: RACE, band-27, varying $C$, *gpu-scs-cm-shfl*. $p = 5$, Mind the scale of the $y$-axis, recall that $N_{\text{tpw}} = C \cdot N_{\text{tpr}}$.

While plotting RACE performance over matrix shows the problem of level groups very well, it also shows the implications on large matrix performance. After all, the influence of the ramp up in regards to large matrix performance is what matters in the end. This also shows why we previously fixated on $C = 8$ for all the tests. In the majority of cases it simply performs best for all GPUs. Theoretically this could change if a matrix' $N_{\text{nzr}}$ is divisible by $N_{\text{tpr}}$, but this would be an unreasonable assumption for an arbitrary matrix.

#### 4.4.2.5 Safety Factor Optimization

We mentioned previously that we choose RACE's cache size parameter to be twice the size of the actual LLC size. This is because RACE internally adjusts the available cache by a safety factor 0.5×. It accounts for the vectors and other overhead which may occupy the cache. If we choose to entirely compensate for this factor, we assume that no overhead occurs. That may not be true in practice, so we show an analysis which cache size parameter performs best.

In our previous graph from Figure 4.4, we can see that performance drops do not occur at precisely the cache size. This is especially the case for the A100 and GH200. Thus, the ideal cache size parameter may vary in practice.

In order to determine whether it is related to the cache size parameter, we show a plot with increasing `--cache-size` parameter in Figure 4.8. The L40 performs best when the cache size parameter is chosen to be exactly double the physical size. We refer to this as *default safety-adjusted*, and it suggests that the internal safety factor is unnecessary. This is not the case for the RX 6900, as it neither performs best when specifying the real physical cache size, nor when overspecifying it to 2×. So RACE's internal safety factor is not entirely unreasonable, but in the case of the RX 6900 it is certainly not precise. Of course this implies that all our previous results for the RX 6900 are not ideal. However as the difference is within 10%, it should not change any of the conclusions that we have drawn so far.

The best fitting cache size parameters are summarized in Table 4.4, except for the A100 and GH200, which do not show a peak in the graph. The graph actually suggests that it would be better

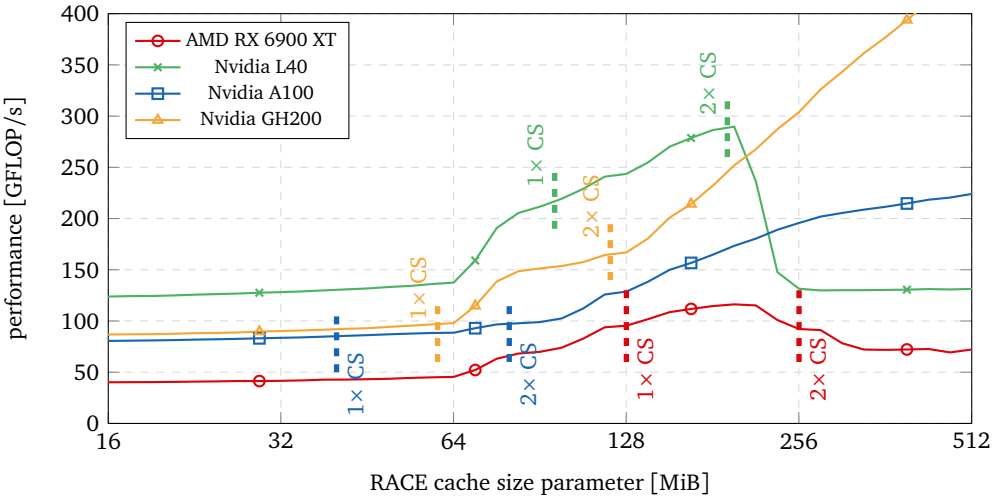



**Figure 4.8** – MPK-HIP: RACE, ML_Geer, $C = 8$, *gpu-scs-cm-shfl*, $p = 5$, varying `--cache-size`. Other matrices experience different graphs, but the peaks occur at the same cache size.

| GPU | $CS_{phy}$ | $P_{phy}$ | $CS_{dsa}$ | $P_{dsa}$ | $CS_{bsa}$ | $P_{bsa}$ |
|---|---|---|---|---|---|---|
| AMD RX 6900 XT | 128 | 95.3 | 256 | 92.1 | 197 | 116.2 |
| Nvidia L40 | 96 | 211.6 | 192 | 289.7 | 192 | 289.7 |
| Nvidia A100 | 40 | 85.4 | 80 | 97.7 | N/A | N/A |
| Nvidia GH200 | 60 | 96.6 | 120 | 125.8 | N/A | N/A |

**Table 4.4** – RACE cache size (CS) in MiB set to physical (`phy`), default 2× safety-adjusted (`dsa`), and best safety-adjusted (`bsa`). $P$ is the belonging performance figure in GFLOP/s. A100 and GH200 do not have a best safety-adjusted performance, as they do not ramp up fully.

to increase the cache size parameter beyond 512 MiB. That, of course, is not a good solution as a bigger parameter will eventually result in one single level group, which is then equivalent to a classic SpMV. For the A100 and GH200 we have already observed poor RACE performance in Figure 4.6. Both of these observations confirm that RACE makes performance worse instead of improving it, at least on those GPUs.

### 4.4.3 Final Comparison

To conclude our analysis, we ran benchmarks with the identical matrices like the authors of [1] in their final performance evaluation. However we omitted the *Lynx1151* matrix due to being too large for the RX 6900's DRAM. We also omitted the *Anderson-600* as it was not available to us. The majority originates from the SuiteSparse collection [31]. *Topi-real-256-256-256* and *Graphene-8192-8192* were generated with ScaMaC [30] and *Lynx649* originates from [21]. The full list is shown in Table 4.3.

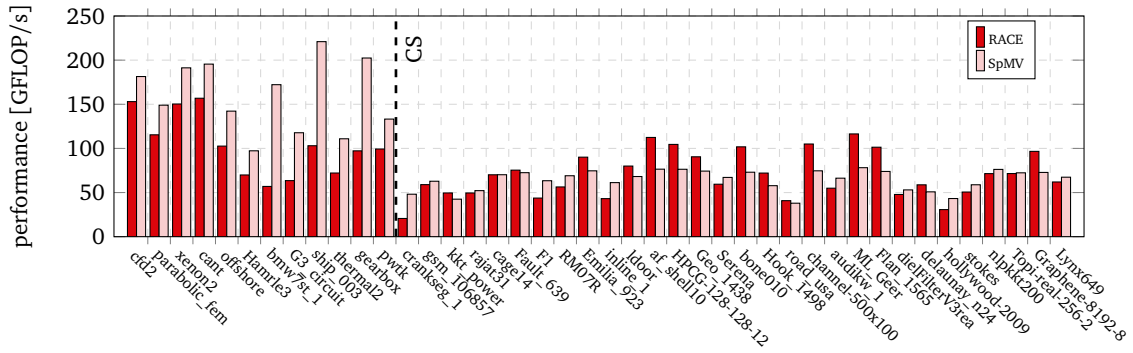
In Figure 4.9 we compared the performance of the RACE MPK against the SpMV MPK, both running on GPU. Strictly speaking, for out-of-cache matrices we experienced no discernible difference between a single SpMV ($p = 1$) run and an MPK ($p > 1$) run with SpMV. Because MPK-HIP always performs the SpMV and RACE MPKs in a single benchmark run, these are shown in the plots. Providing a better chance to the A100 and GH200, we decided to test them at $p = 2$ instead of $p = 5$. All other parameters were kept fixed, and RACE cache size was set to `auto`[11].

As expected from previous results, there is no speedup measurable with RACE for the A100 and GH200. The speedup is better at the measured $p = 2$ compared to $p = 5$, but the performance still remains consistently worse[12]. Both GPUs are more suitable for lower matrix sizes. The reason for that is inherent, since the smaller the matrix, the fewer level groups are necessary for a RACE MPK. If the matrix becomes small enough, RACE will eventually use a single level group for its entire MPK. Thus there will no difference to the SpMV MPK anymore.

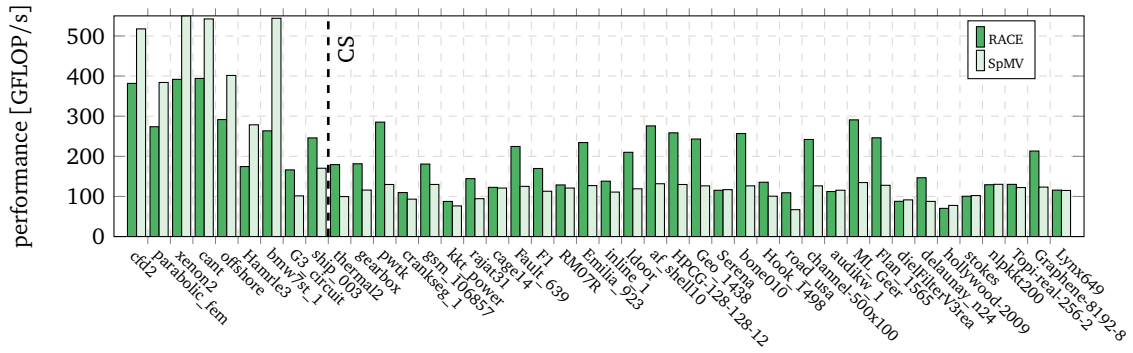
As shown in Section 4.4.2.4, good ramp behavior up is crucial for getting good performance. At least with the current implementation, the A100 and GH200 are unfortunately unable to ramp up fast enough, so good performance cannot be expected. A larger cache size is a remedy to bad ramp up behavior, because that increases level group size and reduce negative impacts on ramp up. Both latter GPUs are the ones with the least amount of cache, which certainly does not help. A good example is the RX 6900: It also has rather bad ramp up behavior, but its cache is the largest of all GPUs. Therefore it is able to get at least a 1.4× speedup for matrices, which partition well like *ML_Geer* and *af_shell10*.

[11] We implemented `auto` according to Figure 4.8 to choose double the available cache size except for the RX 6900, which is overridden to 192 MiB.

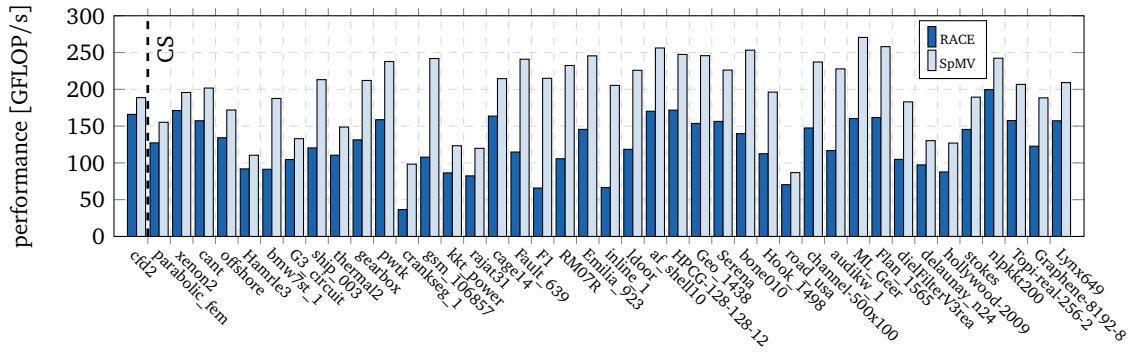
[12] Performance (A100 and GH200) for $p = 5$ is not shown in the graphs.



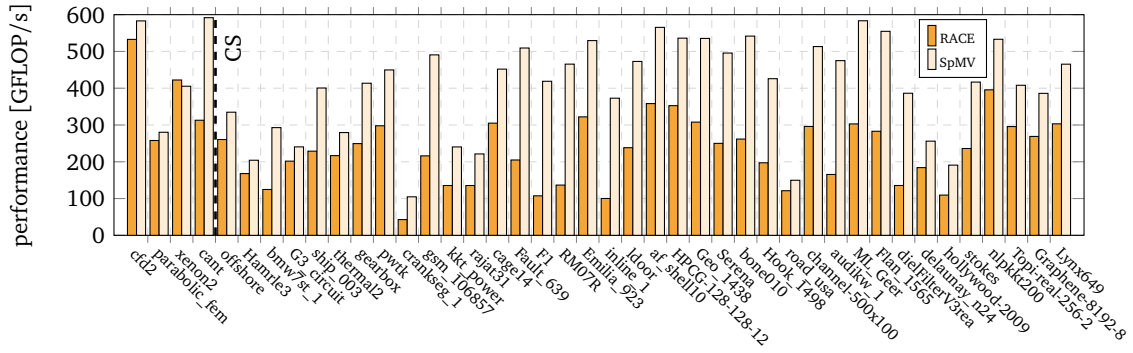

**(a)** AMD RX 6900 XT.




**(b)** Nvidia L40.




**(c)** Nvidia A100.




**(d)** Nvidia GH200.

**Figure 4.9** – MPK-HIP: RACE vs. SpMV, all matrices, $C = 8$, *gpu-scs-cm-shfl*. (a), (b): $p = 5$, (c), (d): $p = 2$. Matrices are sorted ascending in size, left of the dashed black line fit in cache.

The clear outlier is the L40. With minor exceptions it performs consistently at least as good as with the RACE MPK compared to the SpMV MPK. In many cases it actually performs better, and under best cases it reaches a speedup of 2.1×. As for the RX 6900, better performing matrices tend to be of "more diagonal" shape (e.g., *ML_Geer* and *af_shell10*).

Finally, the performance can be summarized as following:

- **Ramp up is crucial:** A GPU may have very fast cache, but if it is not possible to utilize the cache's bandwidth under short loads, our MPK implementation will not perform well.
- **Cache size matters:** The RX 6900 can be considered to be the weakest of all the tested GPUs. nonetheless it is able to get speedup for some matrices, because its large cache can offset its bad ramp up.
- **Speedup is possible:** We have seen for the L40 that substantial performance increase is possible. If the GPU can ramp up fast enough and the cache is sufficiently large, the cache outperforms the DRAM.

In terms of bandwidth, there is additional headroom for improvement. Even on the well performing L40, it typically peaks 30% or less below the roofline. Table 4.5 shows the speedup with RACE when going by the roofline from (4.8). It must be emphasized that no overhead is included and thus the table does not show a drop after a certain power. In real world circumstances the values will decline again, as shown, e.g., in Figure 3.7.

The values in the table also demonstrate that even under best circumstances the GH200 is limited to reach a speedup of $\sim 2\times$. In contrast, the L40 could get as fast as $\sim 4\times$. If it is possible to eliminate or vastly reduce the overhead, the GH200 should certainly profit, but the L40 would likely benefit the most from it. This is of course due to the high discrepancy between its DRAM and LLC bandwidth. Because the A100 and GH200 use much faster HBM DRAM, this ratio is much lower, and therefore the possible speedups.

## 4.5 gpu-small-kernels

As a last measure to exclude errors from ramp up analysis, we used the *gpu-small-kernels* benchmark [9] to cross check its results against ours. *gpu-small-kernels* is a benchmark with primary purpose of measuring ramp up behavior for cache blocking algorithms. It works by running a repeated *scale* stream on GPU over an increasing amount of data. The larger the stream becomes, the fewer times it is executed. Conceptually this is very similar to what RACE does, although *gpu-small-kernels* will stop decreasing the number of kernel calls at a data size of ~8 MiB. This limited decrease ensures good measurement accuracy while keeping the benchmark runtime reasonable.

| GPU | $s_{p=1}$ | $s_{p=2}$ | $s_{p=3}$ | $s_{p=4}$ | $s_{p=5}$ | $s_{p=6}$ | $s_{p=7}$ | $s_{p=8}$ | $s_{p=9}$ | $s_{p=10}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| AMD RX 6900 XT | 1.00 | 1.60 | 1.99 | 2.28 | 2.49 | 2.65 | 2.79 | 2.89 | 2.98 | 3.06 |
| Nvidia L40 | 1.00 | 1.73 | 2.28 | 2.72 | 3.07 | 3.36 | 3.60 | 3.80 | 3.98 | 4.13 |
| Nvidia A100 | 1.00 | 1.50 | 1.80 | 2.00 | 2.14 | 2.25 | 2.33 | 2.40 | 2.45 | 2.50 |
| Nvidia GH200 | 1.00 | 1.43 | 1.67 | 1.82 | 1.92 | 2.00 | 2.06 | 2.11 | 2.14 | 2.18 |

**Table 4.5** – Theoretically possible speedup $s$ by roofline from (4.8). Unlike measurements there is no drop after a certain power since overhead is not taken into account.

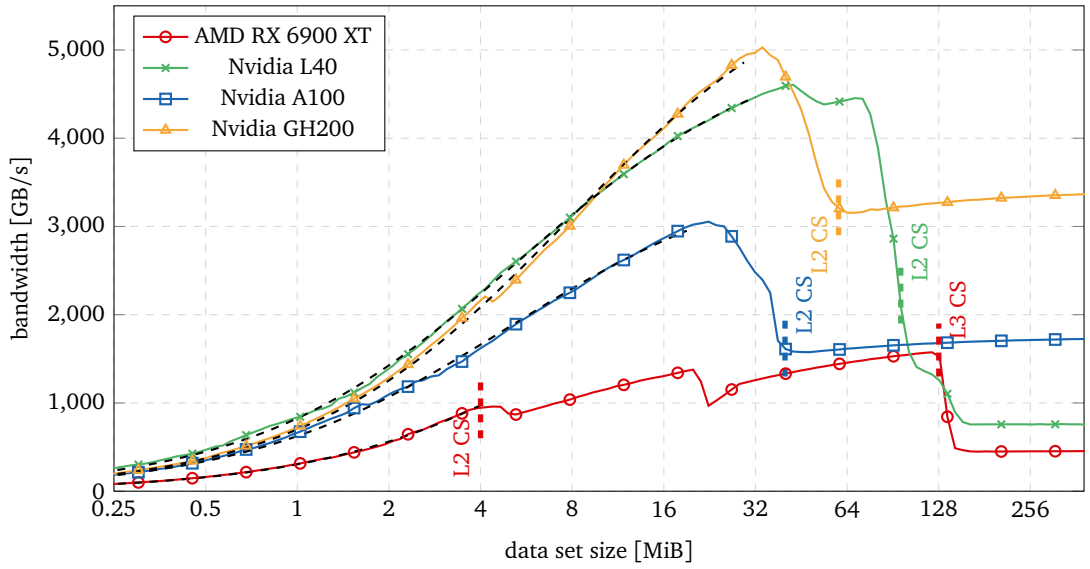



**Figure 4.10** – gpu-small-kernels bandwidth, `-graph` enabled, 256 threads per block. Colored dashed lines mark the real cache size. Black dashed lines mark the fitted curves.

Figure 4.10 shows the bandwidth scaling for an increasing data set size. A similar trend is seen, as previously observed when measuring the SpMV scaling in Figure 4.4. Note that the previous SpMV benchmark measures GFLOP/s and not GB/s, and *gpu-small-kernels* uses an exact data intensity of 16 bytes per FLOP. As the kernel consists only of a single scale operation, it is expected to perform more consistently than the vastly more complex multi-threaded SELL-$C$-$\sigma$ SpMV kernel. This is exactly what can be seen. One of the main differences being that the SpMV kernel is a lot more read intensive, which may result in different cache evictions. In addition the SpMV kernel experiences branch divergence for $N_{\text{nzr}}$ which are not divisible by $N_{\text{tpr}}$.

Both the RX 6900 and L40 show a drastic drop in bandwidth at their LLC boundary. As they did for SpMV scaling, the A100 and GH200 already experience a drop before the data size reached the physical cache size. Interestingly, the RX 6900 also shows the same drop when going over its L2 cache boundary, which in both cases is not exactly at the expected 4 MiB. It also shows the same drop at around 20 MiB to 24 MiB, which is not a known cache boundary, but it at least shows that this drop is not solely present on the SpMV kernel.

The authors of *gpu-small-kernels* also provide a formula to model the bandwidth ramp up [9]. This model assumes that every kernel call has a fixed time of overhead $T_{\text{ovh}}$, while the rest executes at full bandwidth $B_{\text{full}}$:

$$B_{\text{eff}} = \frac{Q}{T_{\text{ovh}} + \frac{Q}{B_{\text{full}}}} \tag{4.12}$$

The two unknowns $T_{\text{ovh}}$ and $B_{\text{full}}$ can then be fitted onto the measured results. Our fitted values can be seen in Table 4.6, along with the resulting curves visualized in Figure 4.10 as dashed black lines. Graphically, they appear to describe the ramp up very well. For the RX 6900 and L40 the fitted cache bandwidth is also reasonably accurate. However the RX 6900 curve is only fitted inside the L2 cache size window, and its cache speed does not line up with *gpu-l2-cache*. If the estimated full

cache bandwidth for the A100 and GH200 is correct, it would imply that their true cache bandwidth would be difficult to achieve, at least without making the kernel vastly more complex.

If each kernel call indeed has a fixed time cost, we can make the following comparison to a real measurement: For example, calculating $p = 5$ of *ML_Geer* on the RX 6900 causes RACE to create 92 level groups, thus causing 460 kernel calls. The kernel manages to run 104 times per second, which means a total number of ∼48000 kernel calls per second. That means that each kernel runs for ∼20 µs. While this is not negligible compared to the 3.1 µs of overhead, it does not explain why already well performing matrices do not reach the roofline from (4.8).

Because of the discussed uncertainties, we do not consider this model to be generally applicable. The in-cache bandwidth for the A100 and GH200 should be much higher, and their poor RACE performance cannot be explained with a very low overhead of 1.3 µs. Drawing a final comparison between *gpu-small-kernels* to the previous benchmarks is difficult. They measure different behaviors, but in the end it confirms that the SpMV's ramp up is not abnormal. In contrast, it can be stated that *BabelStream* (see Figure 4.1) may not the best choice for measuring ramp up, as it only measures single kernel calls and not batched kernel calls.

| GPU | $T_{\text{ovh}}$ | $B_{\text{full}}$ |
|---|---|---|
| AMD RX 6900 XT | 3.15 µs | 3650 GB/s |
| Nvidia L40 | 1.06 µs | 5205 GB/s |
| Nvidia A100 | 1.39 µs | 3721 GB/s |
| Nvidia GH200 | 1.32 µs | 3393 GB/s |

**Table 4.6** – gpu-small-kernels: fitted values for effective bandwidth model from (4.12).

# 5 CONCLUSION

## 5.1 Summary

In this thesis we have described how to run cache-blocked MPKs on GPU using RACE. We started by describing the context of MPKs in their application for Krylov Subspace Methods and the possible speedup for CPUs using RACE. We developed our GPU benchmark MPK-HIP with an approximate roofline model and the knowledge of common matrix formats. This benchmark takes advantage of the GPU optimized matrix format SELL-$C$-$\sigma$ to form an SpMV kernel that is able to operate on a subset of the matrix' rows. RACE is then used to run this kernel in a cache-optimized order to create the GPU RACE MPK. Finally, we presented a roofline for the GPU RACE MPK, showed the measurements results, and explained their deviance.

Under best circumstances our benchmark can achieve a 2× speedup on the L40 and 1.4× on the RX 6900. Unfortunately the A100 and GH200 fall behind and are unable to run faster than a classic SpMV MPK. We discovered that ramp up behavior and cache play a critical role with regard to performance. If the cache is small and ramp up is poor, performance will suffer. All shown results are of course constricted by the "single kernel call per level group" implementation. According to our roofline model for RACE good speedup should be possible for all GPUs. Of course this model assumes that it is possible to reduce the current overhead of the implementation. Although performance is already good in selected cases, a solution running well across a broad range of hardware would make MPKs as promising as the RACE based CPU MPK.

## 5.2 Future Work

First and foremost, assuming that ramp up behavior will not improve on future GPUs, the only alternative is to compensate its negative effects with larger caches. One example for such a GPU is the AMD MI300X which was unfortunately not available to us at the time. The only alternative is to either reduce the ramp up overhead with better hardware, or to implement a smarter MPK that works around the slow ramp up. While not necessarily "better", we did not test GPUs from Intel. It would be insightful to run MPK-HIP on these to see if they behave similarly, but since HIP does not run on Intel GPUs that would require a rewrite in SYCL.

Optimizing the current version of MPK-HIP could be possibly done by a similar approach as the creators of RACE originally did via "Point-to-Point Synchronization". The concept is to relax the synchronization barrier after each level group. This barrier exists implicitly on our GPU implementation as kernel calls, but it is of course possible to further divide these kernel calls into smaller kernels

that run asynchronously. In theory this should yield better concurrency on GPU, but it would also increase the total kernel call count. Is is therefore not certain whether it would work.

To conclude, there are of course more possibilities for further investigation. However, we would consider these lower priority as they do not decide over the viability of GPU MPKs. For example, we currently do not have automatic parameter tuning for $C$, $p$, or `--cache-size` implemented. Having static tuning rules would be preferable, as they do not increase preprocessing cost as much as measurement based tuning. Tuning `--cache-size` would implicitly also include adjustment of RACE's safety factor. Currently RACE does not know of the matrix size increase from SELL-$C$-$\sigma$, which we built on top, and this could be relevant for matrices with high fill-in ratio. Although we determined that all GPUs performed best with $C = 8$, this depends to a certain extent on the matrix and is one more tuning option.

# LIST OF ACRONYMS

| | |
|---|---|
| **BFS** | Breadth-First Search |
| **CA-GMRES** | Communication-Avoiding GMRES |
| **CG** | Conjugate Gradient |
| **CPU** | Central Processing Unit |
| **CRS** | Compressed Row Storage |
| **CUDA** | Compute Unified Device Architecture |
| **DP** | Double Precision |
| **DRAM** | Dynamic Random Access Memory |
| **GMRES** | Generalized Minimal Residual Method |
| **GPU** | Graphics Processing Unit |
| **HIP** | Heterogeneous Interface for Portability |
| **MGS** | Modified Gram-Schmidt |
| **KSM** | Krylov Subspace Method |
| **LLC** | Last Level Cache |
| **LRU** | Least Recently Used |
| **MPK** | Matrix Power Kernel |
| **NUMA** | Non-Uniform Memory Access |
| **RACE** | Recursive Algebraic Coloring Engine |
| **SELL** | Sliced ELLPACK |
| **SIMD** | Single Instruction Multiple Data |
| **SIMT** | Single Instruction Multiple Thread |
| **SIMT** | Single Instruction Multiple Threads |
| **SIMD** | Single Instruction Multiple Data |
| **SP** | Single Precision |
| **SpMV** | Sparse Matrix-Vector |
| **TSQR** | Tall Skinny QR |

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF LISTINGS

# REFERENCES


[1] Christie Alappat et al. "Level-Based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication." In: *IEEE Transactions on Parallel and Distributed Systems* 34.2 (2022), pp. 581–597.

[2] *AMD EPYC 9654P*. URL: https://www.amd.com/en/product/12251 (visited on 04/13/2024).

[3] Zhaojun Bai, Dan Hu, and Lothar Reichel. "A Newton basis GMRES implementation." In: *IMA Journal of Numerical Analysis* 14.4 (1994), pp. 563–581.

[4] Carlos Carvalho. "The gap between processor and memory speeds." In: *Proc. of IEEE International Conference on Control and Automation*. Vol. 5000. 10000. 2002, p. 15000.

[5] Tom Deakin et al. "GPU-STREAM v2. 0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models." In: *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^ 3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers 31*. Springer. 2016, pp. 489–507.

[6] James Demmel et al. "Avoiding communication in sparse matrix computations." In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536305.

[7] Adam Dziekonski, Adam Lamecki, and Michal Mrozowski. "A memory efficient and fast sparse matrix vector product on a GPU." In: *Progress in electromagnetics research* 116 (2011), pp. 49–63.

[8] Michael Garland et al. "Parallel Computing Experiences with CUDA." In: *IEEE Micro* 28.4 (2008), pp. 13–27. DOI: 10.1109/MM.2008.57.

[9] *GPU benchmarks*. URL: https://github.com/te42kyfo/gpu-benches (visited on 03/27/2024).

[10] Magnus Rudolph Hestenes, Eduard Stiefel, et al. *Methods of conjugate gradients for solving linear systems*. Vol. 49. 1. NBS Washington, DC, 1952.

[11] *HIP Documentation*. URL: https://rocm.docs.amd.com/projects/HIP/en/latest/ (visited on 04/14/2024).

[12] Mark Hoemmen. *Communication-avoiding Krylov subspace methods*. University of California, Berkeley, 2010.

[13] Wayne D Joubert and Graham F Carey. "Parallelizable restarted iterative methods for non-symmetric linear systems. Part I: Theory." In: *International journal of computer mathematics* 44.1-4 (1992), pp. 243–267.

[14] Shoaib Kamil et al. "Implicit and explicit optimizations for stencil computations." In: *Proceedings of the 2006 workshop on Memory system performance and correctness*. 2006, pp. 51–60.

[15] Konstantinos I. Karantasis et al. "Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction." In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 921–932. DOI: 10.1109/SC.2014.80.

[16] Kamran Karimi, Neil G Dickson, and Firas Hamze. "A performance comparison of CUDA and OpenCL." In: *arXiv preprint arXiv:1005.2581* (2010).

[17] Moritz Kreutzer et al. "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units." In: *SIAM Journal on Scientific Computing* 36.5 (2014), pp. C401–C423.

[18] Moritz Kreutzer et al. "Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation." In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE. 2012, pp. 1696–1702.

[19] Monica D Lam, Edward E Rothberg, and Michael E Wolf. "The cache performance and optimizations of blocked algorithms." In: *ACM SIGOPS Operating Systems Review* 25.Special Issue (1991), pp. 63–74.

[20] Cornelius Lanczos. "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators." In: (1950).

[21] Johannes Langguth et al. "Scalable heterogeneous CPU-GPU computations for unstructured tetrahedral meshes." In: *IEEE Micro* 35.4 (2015), pp. 6–15.

[22] Henry Lucas. "Performance Evaluation and Monitoring." In: *ACM Comput. Surv.* 3.3 (Sept. 1971), pp. 79–91. ISSN: 0360-0300. DOI: 10.1145/356589.356590. URL: https://doi.org/10.1145/356589.356590.

[23] Nihar R Mahapatra and Balakrishna V Venkatrao. "The processor-memory bottleneck: problems and solutions." In: *Xrds* 5.3es (1999), 2–es.

[24] Maryam MehriDehnavi et al. "Communication-avoiding Krylov techniques on graphic processing units." In: *IEEE transactions on magnetics* 49.5 (2013), pp. 1749–1752.

[25] Marghoob Mohiyuddin et al. "Minimizing communication in sparse matrix solvers." In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009, pp. 1–12.

[26] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. "Automatically tuning sparse matrix-vector multiplication for GPU architectures." In: *High Performance Embedded Architectures and Compilers: 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings 5*. Springer. 2010, pp. 111–125.

[27] *NVIDIA H200 Tensor Core GPU*. URL: https://nvdam.widen.net/s/nb5zzzsjdf/hpc-datasheet-sc23-h200-datasheet-3002446 (visited on 04/13/2024).

[28] Georg Ofenbeck et al. "Applying the roofline model." In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2014, pp. 76–85.

[29] Youcef Saad and Martin H Schultz. "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems." In: *SIAM Journal on scientific and statistical computing* 7.3 (1986), pp. 856–869.

[30] *ScaMaC - a Scalable Matrix Collection*. URL: https://bitbucket.org/essex/matrixcollection/src/master/ (visited on 04/26/2024).

[31] *SuiteSparse Matrix Collection*. URL: https://sparse.tamu.edu/ (visited on 04/15/2024).

[32] Francisco Vázquez, José-Jesús Fernández, and Ester M Garzón. "A new approach for sparse matrix vector product on NVIDIA GPUs." In: *Concurrency and Computation: Practice and Experience* 23.8 (2011), pp. 815–826.

[33] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://doi.org/10.1145/1498765.1498785.

[34] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. University of California, Berkeley, 2008.

[35] Markus Wittmann et al. "Multicore performance engineering of sparse triangular solves using a modified roofline model." In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2018, pp. 233–241.

[36] Wm A Wulf and Sally A McKee. "Hitting the memory wall: Implications of the obvious." In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[37] Ichitaro Yamazaki et al. "Improving the performance of CA-GMRES on multicores with multiple GPUs." In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 382–391.