Converting CUDA programs to run on AMD GPUs

Master's Thesis

Victor Anderssén

Department of Information Technologies
Faculty of Science and Engineering
Åbo Akademi University
Supervisor: Jan Westerholm
Vasa April 23, 2024
Student Number: 2101175

Abstract

This Master's thesis outlines the conversion process of CUDA (Compute Unified Device Architecture) source code, originally designed for Nvidia hardware, to code compatible with both Nvidia and Advanced Micro Devices (AMD) hardware. The conversion is facilitated by AMD's Heterogeneous-Computing Interface for Portability (HIP).

Parallel computing using Graphics Processing Units (GPUs) has been a long-standing practice, leveraging GPUs' parallel capabilities, software support, and speed. Nvidia has historically dominated the supercomputing market for GPUs. However, with the emergence of Advanced Micro Devices (AMD) as a competitor, a viable alternative for supercomputers now exists. Notably, the LUMI supercomputer in Finland incorporates AMD GPUs in its GPU partition.

This Master's thesis addresses the challenge of converting pre-existing CUDA code, along with associated libraries, into HIP code, thereby enabling execution on hardware from both manufacturers. This HIP conversion empowers developers to run their legacy CUDA programs seamlessly on Nvidia or AMD hardware. A general method for the 'hipification' of external libraries and own code was developed.

The general method was tested on a Quasi-Minimal Residual method (QMR) solver. The test data that the QMR solver is run with is from DREAM (Disruption and Runaway Electron Analysis Model). Quadruple precision was used in order to achieve convergence in the QMR solver. The runtime of the QMR program was measured for different GPUs resulting in comparable runtimes for both AMD and Nvidia GPUs.

Keywords: *AMD, C, CUDA, GPU, HPC, HIP, Nvidia, QMR, Quadruple Precision, ROCm, SLEEF.*

Acknowledgements

Firstly, I would like to express my gratitude to Åbo Akademi for providing me with the opportunity to undertake my Master's Thesis work with them. In particular, I extend my appreciation to my supervisor, Jan Westerholm, who entrusted me with this thesis, allowing me the freedom to approach the task according to my own discretion and providing invaluable expertise and assistance throughout the process. Additionally, I would also like to express a thank you to Christel Björkstrand and Malin Johansson who have helped me with the writing of this thesis.

Lastly, I would like to express my gratitude to every teacher and personnel at Åbo Akademi who have played a part in my academic journey. Their dedication, support, and guidance have been invaluable in shaping my educational experience and contributing to the completion of my Master's Thesis.

Abbreviations

AMD (Advanced Micro Devices)

API (Application Programming Interface)

C/C++ (C/C++ programming language)

CPU (Central processing unit)

CUDA (Compute Unified Device Architecture)

GPU (Graphics processing unit)

HIP (Heterogeneous-Computing Interface for Portability)

LLVM (Low level virtual machine)

NVCC (Nvidia CUDA compiler)

QMR (Quasi-Minimal Residual method)

RAM (Random Access Memory)

SIMD (Single Instruction / Multiple Data)

Contents

Appendices

Chapter 1

Introduction

Everyday mathematical algorithms are in many cases solved with supercomputers. It is of paramount importance that these algorithms exhibit optimal swiftness and precision, as fast programs not only conserve valuable time but also mitigate financial expenditures. Nevertheless, harnessing the full spectrum of available computational capabilities can be challenging. This task requires a deep understanding of how to do calculations at the same time (in parallel), as well as a nuanced expertise in the domain-specific intricacies associated with Graphics Processing Units (GPUs). The utilization of pre-established software libraries for problem resolution constitutes a prevalent and pragmatic approach, particularly in the domain of computational mathematics and physics. There is no rationale in re-inventing the wheel by rewriting existing algorithms. Previously, CUDA programs had to be recompiled for the new supercomputers. Nowadays, the hardware support in new supercomputers may switch between CUDA, HIP, and back to CUDA.

Parallel programming constitutes a programming paradigm in which a given program is concurrently executed on multiple Central Processing Units (CPUs) or multiple GPUs. The predominant benefit inherent to parallel programming is the substantial increase in computational speed. When a program exhibits parallelizability, the utilization of parallel computing methodologies becomes advantageous. It is noteworthy that a single GPU can execute a considerably greater number of parallel programs in comparison to a CPU containing dozens of CPU cores, demonstrating the inherent performance advantage of GPU-based parallelization.

1.1 Objectives

The thesis aims to facilitate the execution of pre-existing Compute Unified Device Architecture (CUDA) code with external libraries on Advanced Micro Devices (AMD) GPUs. AMD has devised a conversion process that transforms CUDA code into Heterogeneous-Computing Interface for Portability (HIP) code that is compatible with both AMD and Nvidia hardware. The primary goal is to empower CUDA code developers to execute their programs with external libraries on either AMD or Nvidia hardware. By employing the general conversion method outlined in this thesis, it becomes viable to convert and execute CUDA programs alongside external libraries. This addresses a widespread challenge faced by researchers and programmers when new supercomputers are constructed with hardware configurations differing from those they are accustomed to programming for directly. The generalized conversion process, specifically described in the methods chapter, involves the transformation of CUDA code present in both an external library and the source code into HIP code.

1.2 Structure of the thesis

The thesis starts with a literature review, providing background theory on GPUs, parallel computing, and the significance of floating-point precision. The central emphasis of this work is directed toward the orchestration of code execution employing external CUDA libraries on heterogeneous hardware configurations. The following section presents and analyzes the research findings, with an accompanying commentary.

Chapter 2

Background

In the worldwide consumer GPU market, historically characterized by the prominent duopoly of Nvidia and AMD, Intel has commenced its entry into this arena as of the year 2022 [1]. Notably, within the domain of supercomputing, both Nvidia and AMD GPUs are widely deployed. Nevertheless, there is a discernible trend wherein an increasing number of supercomputing facilities are opting for AMD hardware, encompassing both CPUs and GPUs.

Bailey and Borwein's research on "High-Precision Computation and Mathematical Physics" [2] highlights the crucial need for increased mathematical precision in scientific fields such as quantum field theory and supernova simulations. Their work underscores the significance of enhanced precision for advancing computational physics. They have demonstrated the necessity for quadruple precision or higher in various illustrative instances. A common challenge faced in these domains is the requirement to utilize third-party libraries to achieve elevated levels of numerical precision. This thesis explores a gap in current research about combining third-party CUDA libraries with existing source code. It focuses on the need for converting and ensuring compatibility in the process.

In the upcoming Background chapters, the following topics will be presented: CUDA and HIP history, the LUMI consortium and LUMI supercomputer, GPU programming, the GPU memory model, AMD ROCm, Makefiles, Hipify, and the quadruple-precision floating-point format using the external library SLEEF.

2.1 CUDA and HIP history

CUDA has been the main software development environment in which to write GPU programs up until AMD released its Heterogeneous-Computing Interface for Portability library November 14, 2016 [3]. AMD is now a major competitor to Nvidia. Many new supercomputers are using AMD GPUs; hence, there is a need to convert existing CUDA source code that previously only ran on Nvidia GPUs to run on AMD GPUs through "hipifying" the source code with HIP. The HIP API is very similar to the CUDA API, as can be seen in Table 2.1.

CUDA	HIP
`#include "cuda.h"`	`#include "hip/hip_runtime.h"`
`cudaMalloc(&d_x, N*sizeof(double));`	`hipMalloc(&d_x, N*sizeof(double));`
`cudaMemcpy(d_x,x,N*sizeof(double), cudaMemcpyHostToDevice);`	`hipMemcpy(d_x,x,N*sizeof(double), hipMemcpyHostToDevice);`
`cudaDeviceSynchronize();`	`hipDeviceSynchronize();`

Table 2.1 Differences between CUDA and HIP API [4]

2.2 LUMI consortium and supercomputer

The LUMI supercomputer is hosted by the LUMI consortium, consisting of 11 European countries. The LUMI consortium is part of the European High Performance Computing Joint Undertaking (EuroHPC JU) whose purpose is to coordinate and pool compute resources in order to make Europe a world leader when it comes to supercomputing. [5] [6]

The GPU partition LUMI-G consists of 2978 nodes, each node with one 64 core AMD EPYC "Trento" CPU and four AMD MI250X GPUs. The CPU partition in LUMI is called LUMI-C. It has 2048 nodes with two AMD EPYC 7763 CPUs per node for a total of 262,144 CPU cores. As of November 2023 LUMI is the fifth fastest supercomputer in the world. The energy consumption of LUMI is 7.1 MW and noteworthy is that LUMI is using 100% hydro-powered energy. Waste heat is used for district heating in the nearby area in Kajaani. The waste heat produced accounts for 20% of the total district heating. [7] [8] [9] [10]

$$2978_{\text{GPU Nodes}} \times 4_{\text{AMD MI250X GPUs}} \times 2_{\text{Dies}} = 23,824 \text{ GPU dies total} \quad (2.1)$$

Equation 2.1 shows how the total number of GPU dies for the LUMI-G partition is calculated.

Figure 2.1 LUMI Supercomputer

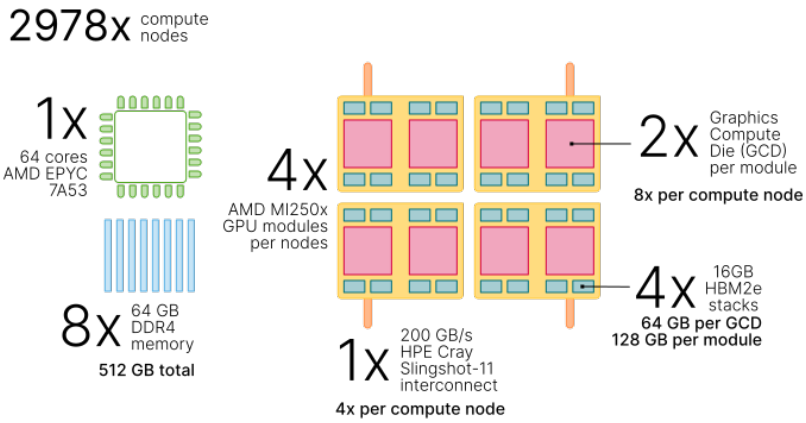

Figure 2.2 Overview of a LUMI-G compute node

2.3 GPU programming

A GPU kernel is the program that executes on the GPU. The difference between a GPU kernel and a CPU C function is that a GPU kernel can spawn orders of magnitude more threads on the GPU than a CPU C function can on the CPU. This makes GPUs a viable choice when it comes to parallel programming. Software designers strive to make programs run fast; time is money. GPUs have taken a central role in parallel computing during the last decade. A GPU can execute thousands of programs or kernels at the same time. Noteworthy is that a GPU executes the spawned kernels in any order. To be able to take advantage of the parallelism in GPUs the program needs to be "loop independent". That is, the result should be independent of the order in which the programs are run.

2.4 Loop independency

The inner part of a loop is a block considered independent when the computation for index value i is independent of the computation for index value $j \neq i$. One method for checking if this holds is to run the loop backwards and check if the results are the same. A matrix multiplication example on the CPU and on the GPU with CUDA can be found in appendix [2].

2.5 GPU Programming Model

A basic GPU programming model looks like this: Design the program to run many independent threads, each operating on small amounts of data. The code executed in each thread could be thought of as the inner code of a loop. Threads are grouped into blocks, and blocks are grouped into a grid of blocks. The threads are executed in groups called waves or warps. Nvidia uses 32 as its warp size and AMD uses 32 or 64 as its wavefront size depending on the chip. [11] [12]

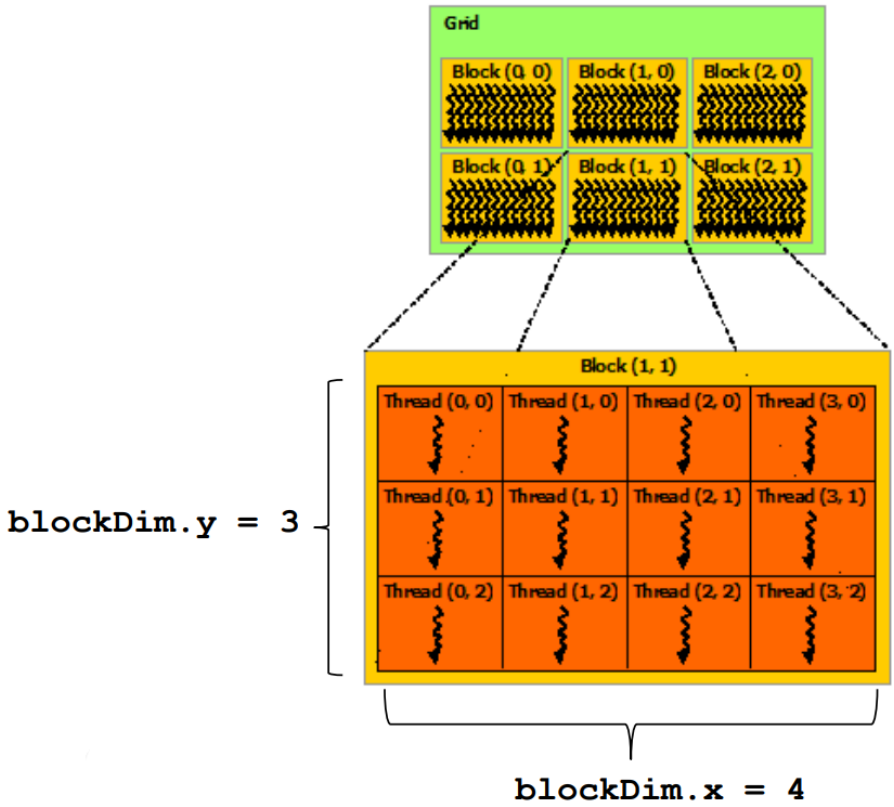

Figure 2.3 GPU programming model

2.6 GPU Memory model

GPUs have their own memories. Data is explicitly transferred from and to the host. The simplest approach is to use only global GPU memory, with the drawback of slower memory access times. Memory is first allocated on the host machine e.g. Random Access Memory (RAM), with *calloc()*. The contents of that memory are then copied over to the GPU memory. The GPU runs the kernel and then some GPU memory containing the result has to be copied back to the host [15]. CUDA unified memory is another option that is a single memory space for both the host CPU and device GPU [13]. For optimization purposes, threads in the same thread block or thread block cluster may access the same shared memory to gain execution speed. Registers only have a lifetime of one thread. Thread Block Clusters were introduced in Nvidia Compute Capability ≥ 9.0.

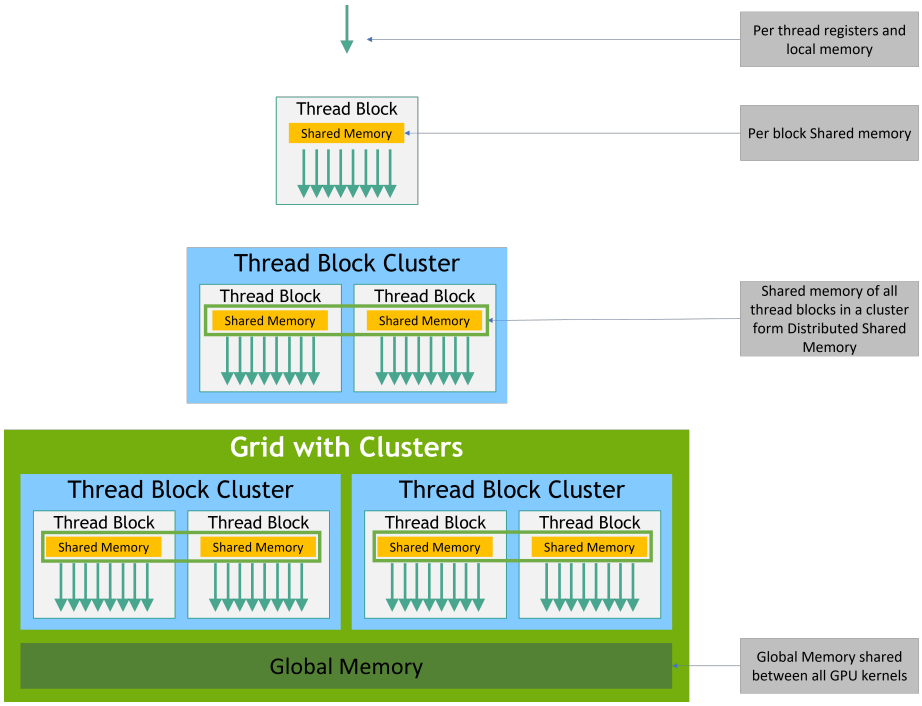

Figure 2.4 CUDA Memory hierarchy

2.7 CUDA

CUDA is Nvidia's API for programming and running programs or kernels on the GPU. It is based on standard C code with a small number of extensions. CUDA was first introduced in 2006 and it is widely used in the field of computer science today. [14]

2.8 HIP

HIP is a C++ dialect designed for porting CUDA programs to portable C++ code. It is AMD's GPU programming environment. It is possible to target both Nvidia and AMD architectures with HIP. Example HIP code that adds a vector can be found in appendix [13]

2.9 AMD ROCm

ROCm is AMD's software stack designed for GPU programming mostly comprised of open-source software. ROCm is powered by HIP. It supports multiple programming models such as OpenMP and OpenCL and frameworks e.g. PyTorch and Tensorflow and libraries such as hipBLAS, hipFFT and hipSPARSE. [15]

2.10 Compiler

A compiler is a program that takes source code (C / C++ / CUDA / HIP) as input and outputs byte code or machine code for the CPU or GPU to execute. Two common compilers are GCC and Clang.

2.11 Makefile

Make is a build system used for compiling and building software projects. A Makefile is a file that includes steps on how the source code should be compiled. It can also keep track of which parts of the code that needs recompiling, shortening the total compile time. The makefile consists of variables and sections. Each section performs a specific part of the compilation process. See Appendix [7] for a complete Makefile.

2.12 Hipify

The conversion from CUDA to HIP can partly be automated using the hipify tools provided by AMD. The tool exists in two flavors. The simpler one is a hipify-perl perl script that performs a find and replace on the project source files. The second tool is hipify-clang which is a more sophisticated tool based on Clang and Low level virtual machine (LLVM). [16]

The hipify tools try to convert CUDA API calls to HIP API calls. Some manual work might be needed to get the HIP program to compile, such as header includes and compiler flags.

2.13 Quadruple-precision floating point format

Normally, when coding with floating point numbers in C, floats with 6 significant digits are used. However, if the program demands more precision, doubles with 15 significant digits are employed. These two data types in C occupy 4 bytes and 8 bytes, respectively. A float128 occupies 16 bytes (128 bits) of memory and has 33 significant digits. [2].

Quadruple-precision layout: IEEE 754 quadruple-precision format includes:

- **Sign Bit (1 bit):** The most significant bit represents the sign of the number, where 0 denotes positive and 1 denotes negative.
- **Exponent (15 bits):** This portion encodes the exponent of the number, allowing a wide range of values.
- **Significand (Mantissa) (112 bits):** The significand contains the fractional part of the number, allowing a high degree of precision.

A quadruple-precision floating-point value can be created by combining three 64-bit components: sign, exponent, and significand. An over simplified version of this can be found in Appedix: [8]. Extraction of these components can be achieved with bit manipulations.

The IEEE 754 standard defines floating-point arithmetic formats, operations, and exception handling. If you want to ensure that float128 in C is IEEE 754 compliant, you need to consider various aspects, including arithmetic operations, trigonometric operations, transcendental operations, and comparison.

Here are some key points to consider for IEEE 754 compliance:

1. **Arithmetic Operations:**

   - Addition ($+$), subtraction ($-$), multiplication ($*$), and division ($/$) should follow the rules specified in IEEE 754, including proper handling of special values such as infinity, NaN (Not a Number), and denormalized numbers.

2. **Trigonometric Operations:**

   - Trigonometric functions like sine, cosine, tangent, etc., should provide accurate enough results according to the IEEE 754 standard.

3. **Transcendental Operations:**

   - Functions like exponentiation (exp()), logarithm (log()), and square root (sqrt()) should adhere to the IEEE 754 specifications, ensuring accuracy and proper handling of special cases.

4. **Comparison:**

   - Comparison operations ($<, <=, ==, \neq, >, >=$) should follow the rules outlined in IEEE 754. This includes the handling of NaN values and ensuring proper rounding modes.

2.14 SLEEF

SLEEF stands for SIMD Library for Evaluating Elementary Functions. This library implements quadruple-precision types and functions in C, adhering to the standard IEEE 754 quadruple-precision floating-point format. This is the library used in the implementation of this thesis for quadruple-precision numbers. In SLEEF, a *Sleef_quad* is defined as *__float128* on the CPU. If run on the GPU, *Sleef_quadx1* is used. [17] [18] [19]

SLEEF API examples

```c
#include <quadmath.h>
#include <stdio.h>
#include <sleef.h>
#include <sleefquad.h>

int main(int argc, char **argv)
{
    printf("\n\tSleef Add example\n");

    // Define two __float128 numbers
    Sleef_quad NumberA = 1.337Q;
    Sleef_quad NumberB = 4.200Q;

    // Add the two __float128 numbers together
    Sleef_quad Sum = Sleef_addq1_u05(NumberA, NumberB);

    // Print the result
    Sleef_printf("\tNumberA: %.40Pg\n", &NumberA);
    Sleef_printf("\tNumberB: %.40Pg\n", &NumberB);
    Sleef_printf("\tSum:\t %.40Pg\n", &Sum);

    return (0);
}
```

Figure 2.5 SLEEF addition example with quadruple presicion numbers. From appendix [9]

The numbers in the output [2.6] are double precision floating-point representations in C. The values are the exact value of their internal binary representation, this will lead to the numbers looking funny due to the limitations of representing real numbers in binary using floats or doubles.

```
Sleef Add example
NumberA: 1.336999999999999999999999999999932207
NumberB: 4.200000000000000000000000000000154074
Sum:     5.537000000000000000000000000000086282
```

Figure 2.6 SLEEF addition example output.

```c
#include <quadmath.h>
#include <stdio.h>
#include <sleefquad.h>

int main(int argc, char **argv)
{
    printf("\n\tSleef Multiply example\n");

    // Define two __float128 numbers
    Sleef_quad NumberA = 1.337Q;
    Sleef_quad NumberB = 4.200Q;

    // Multiply the two __float128 numbers together
    Sleef_quad Sum = Sleef_mulq1_u05(NumberA, NumberB);

    // Print the result
    Sleef_printf("\tNumberA: %.40Pg\n", &NumberA);
    Sleef_printf("\tNumberB: %.40Pg\n", &NumberB);
    Sleef_printf("\tSum:\t %.40Pg\n", &Sum);

    return (0);
}
```

Figure 2.7 SLEEF multiplication example with quadruple presicion numbers. From appendix [10]

```
Sleef Multiply example
NumberA: 1.336999999999999999999999999999932207
NumberB: 4.200000000000000000000000000000154074
Sum:     5.615400000000000000000000000000177494
```

Figure 2.8 SLEEF multiplication example output.

```c
#include <stdio.h>
#include <quadmath.h>

int main(void)
{
    // Define a known _Float128 value
    _Float128 PI = acosq(-1.0Q);

    // Buffer to hold the formatted string
    char Buffer[100]; // Adjust the size accordingly

    // Format the _Float128 value to a string with specific precision
    int DecimalPrecision = 34;
    int Result = quadmath_snprintf(Buffer, sizeof(Buffer), "%.*Qf", DecimalPrecision, PI);

    // Check if the formatting was successful
    if (Result < 0)
    {
        printf("\n\t[ERROR]: Formatting failed!\n");
        return (-1);
    }

    // Display the result with 33 decimal places
    printf("\n\tAsserted: acos(-1.0Q) with %d decimals of precision: %s\n\n",
    DecimalPrecision, Buffer);

    return (0);
}
```

Figure 2.9 SLEEF implementation of printing high precision numbers with quadmath. From appendix [11]

```
Asserted: acos(-1.0Q) with 34 decimals of precision: 3.14159265358979323846264338327950288
```

Figure 2.10 SLEEF printing example output.

Chapter 3

Method

This chapter presents the steps taken to obtain solutions to linear equations of disruptions in fusion reactors of the Tokamak type using the Quasi-Minimal Residual method (QMR). A more general approach to hipify external libraries will also be presented. An introduction to iterative solvers is also introduced.

3.1 Iterative Solvers: General Overview

Iterative solvers are numerical techniques used to find approximate solutions to linear systems of equations through a sequence of repeated steps. These methods are particularly useful for large-scale problems where direct methods might be computationally expensive or impractical for iterative solvers due to the size and the sparsity of the linear system. A direct method often uses a dense matrix as an intermediate step. This makes direct methods impractical for very large matrices, of size 1 million $\times$ 1 million and larger.

Common Characteristics:

1. **Iterative Refinement:** Iterative solvers refine an initial guess for the solution in a step-by-step fashion, continuously improving the approximation.

2. **Krylov Subspace Methods:** Many iterative solvers belong to the family of Krylov subspace methods. The Krylov subspace methods construct a sequence of subspaces derived from powers of the coefficient matrix.

3. **Convergence Criteria:** Iterative solvers typically employ convergence criteria to determine when the approximation is sufficiently accurate. Common criteria include reaching a specified tolerance or a maximum number of iterations.

   Define tolerance:

$$\frac{||Ax - b||}{||b||} < \text{tol e.g. } = 10^{-10} \tag{3.1}$$

   The variables in the equation 3.1 are as follows:

   - A represents the coefficient matrix of the linear system.
   - x is the approximate solution obtained from the iterative solver.
   - b is the right-hand side vector of the linear system.
   - $||\cdot||$ denotes the Euclidean norm, which measures the magnitude of a vector.
   - tol is the tolerance level, representing the maximum allowable relative error.

   The equation 3.1 calculates the relative error between the current approximation Ax and the actual right-hand side b of the linear system. The tolerance tol sets the threshold for this relative error. If the relative error falls below the tolerance level, the iterative solver is considered to have converged to an acceptable solution.

   For example, setting tol $= 10^{-10}$ indicates that the solver should continue iterating until the relative error is less than 10^{-10}, providing a highly accurate approximation to the solution. [20]

4. **Applicability to Large and Sparse Systems:** Iterative methods are particularly well-suited for solving large and sparse linear systems that arise in various scientific, engineering, and computational applications because sparse matrices can be represented and stored efficiently without the use of elements with a value of zero.

Examples of Iterative Solvers:

1. **Conjugate Gradient (CG) method:** One of the best known iterative solvers. Efficient for solving sparse symmetric positive definite matrices. Sparse symmetric positive definite matrices have mostly zero elements, are symmetric across their main diagonal, and have all positive eigenvalues. [20]

2. **GMRES (Generalized Minimal Residual method):** Suitable for nonsymmetric matrices. It was introduced by Y. Saad and M. Schultz in 1986. [21]

3. **BiCGStab (Biconjugate Gradient Stabilized method):** An extension of CG for nonsymmetric matrices with improved stability.

4. **Jacobi Iteration:** A simple iterative method updating each variable based on an average of its neighbors.

5. **SOR (Successive Over-Relaxation) method:** Introduces relaxation parameters to accelerate convergence, particularly for symmetric positive definite matrices.

   The idea behind relaxation parameters is to strike a balance between making significant corrections to the current approximation and ensuring stability in the iterative process. By carefully tuning the relaxation parameter, the iterative solver can converge more rapidly to the solution without sacrificing accuracy or stability.

Applications:

- **Scientific Computing:** Used in simulations, computational physics, and numerical modeling.
- **Engineering:** Applied in structural analysis, fluid dynamics, and finite element methods.
- **Image Processing:** Used for image reconstruction and feature extraction.
- **Optimization:** Employed in optimization algorithms and linear programming.

Conjugate Gradient Method Example

Introduction

The Conjugate Gradient (CG) method is an iterative algorithm for solving symmetric positive definite linear systems. Given a linear system $Ax = b$ where A is symmetric positive definite, the CG method iteratively refines the solution x. [20, 22]

Conjugate Gradient Method

The CG method iteratively refines the solution by updating the current approximation and search direction. The algorithm involves computing coefficients such as α_k and β_{k+1} in each iteration.

Initialization:

x_0 (initial guess)

$r_0 = b - Ax_0$ (initial residual)

$p_0 = r_0$ (initial search direction)

Iteration (for $k = 0, 1, 2, \ldots$):

$$\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k} \text{ (compute step size)}$$

$$x_{k+1} = x_k + \alpha_k p_k \text{ (update solution)}$$

$$r_{k+1} = r_k - \alpha_k A p_k \text{ (update residual)}$$

$$\beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \text{ (compute correction factor)}$$

$$p_{k+1} = r_{k+1} + \beta_{k+1} p_k \text{ (update search direction)}$$

Convergence criterion:
The CG method can be terminated based on individual criteria or their combination. It iteratively refines the solution until it achieves convergence or reaches a specified number of iterations.

$$||r_k|| < \epsilon$$

Alternatively, the change in the solution between iterations can be compared with the predefined tolerance:

$$||x_{k+1} - x_k|| < \epsilon$$

The algorithm can also be terminated if the iteration count exceeds the maximum number of iterations:

$$k \geq \text{max\_iterations}$$

These criteria can be used individually or in combination to determine when to terminate the conjugate gradient method. The CG method iteratively refines the solution until convergence or a specified number of iterations.

3.2 QMR - quasi-minimal residual method

The Quasi-Minimal Residual (QMR) method is an iterative numerical technique used to solve systems of general linear equations. An example of a linear equation can be found in appendix [14]. It is particularly useful for solving large, sparse, and nonsymmetric systems commonly encountered in scientific and engineering simulations. QMR is versatile because it does not assume any specific characteristics about the linear system apart from being nonsingular (invertible) matrices. It is an extension of the Minimal Residual (MINRES) method and is designed to handle problems where other methods like Conjugate Gradient (CG) may not be suitable, as CG requires the matrix to be symmetric. The QMR method in our case necessitates the utilization of quadruple precision arithmetic in order to yield accurate and robust outcomes [23]. The goal is to run our CUDA version of QMR on AMD GPUs. The need for quadruple precision is not general. In our QMR problem with one million variables we are not able to get QMR to converge with tolerance 10^{-22}.

QMR pseudocode example

A is the nonsymmetric matrix.
b is the right-hand side vector.
x_k is the current approximation of the solution.
r_k is the residual vector.
v_k is an auxiliary vector.
p_k and q_k are auxiliary vectors used in the iterations.
α_k, β_k, and ω_k are scalar coefficients.

Appendices 3.1 QMR solver pseudocode.

```
Input: A (nonsymmetric matrix), b (right-hand side vector), x_0 (initial guess),
    tol (tolerance)
Output: x (approximate solution)

Initialize:
r_0 = b - A * x_0
v_0 = r_0
p_0 = 0
q_0 = 0
beta_1 = ||r_0||

for k = 1 to max_iterations do
    alpha_k = (v_{k-1}^T * r_{k-1}) / (v_{k-1}^T * A * v_{k-1})
    p_k = r_{k-1} - alpha_k * A * v_{k-1}
    q_k = A * p_k
    beta_k = ||q_k||

    if beta_k == 0 then
        break // QMR breakdown, handle accordingly
    end if

    omega_k = (p_k^T * q_k) / (q_k^T * q_k)
    x_k = x_{k-1} + alpha_k * v_{k-1} + omega_k * p_k

    r_k = p_k - omega_k * q_k
    v_k = A * r_k

    if ||r_k|| < tol * ||b|| then
        break // Convergence achieved
    end if
end for

Output x_k as the approximate solution
```

3.3 HIP conversion and compilation

The QMR compilation process for HIP has been implemented through a Makefile, involving a series of sequential steps. One step, namely the 'hipify sleef' is conspicuously left commented out in the Makefile. This particular step requires the adaptation of external library source code to a format compatible with the HIP platform; however, for the specific context of this implementation, such a transformation was deemed unnecessary. The build process is initiated with the removal of the prior build directory, followed by the creation of an empty directory. Subsequently, the external library SLEEF is retrieved from its repository on GitHub. An important aspect to address in this procedure involves the potential need to modify the source code of SLEEF to work with HIP. This requires changing the build scripts and Makefiles to use the ROCm HIP compiler (hipcc) instead of the Nvidia CUDA compiler (NVCC). Furthermore, adjustments may be necessary to configure library paths and flags to ensure a smooth integration with ROCm HIP libraries and their dependencies. Unfortunately, this latter aspect requires a certain level of manual effort. In the final step of the process, the Makefile proceeds to compile the source code of the specified target.

3.4 Compiler flags

The hipcc / Clang compiler flag **-ffp-contract=off** was critical for achieving the correct result when executing the QMR program.

FMA and Contractions

This compiler flag is crucial. It defines when the compiler is allowed to form fused floating-point operations e.g. fused multiply-add (FMA). Contractions are compiler optimizations aimed at improving performance by reducing the number of instructions executed. Instead of executing separate instructions for each operation (e.g. multiplication followed by addition), contractions allow the compiler to merge these operations into a single instruction where supported by the underlying hardware. If FMA and contractions is used the operations are performed more quickly and the resulting total runtime is shorter [11]. In certain numerical algorithms, particularly those involving iterative calculations or computations with very small or very large numbers such as QMR, FMA operations might introduce numerical instability or loss of precision and therefore needs to be turned off.

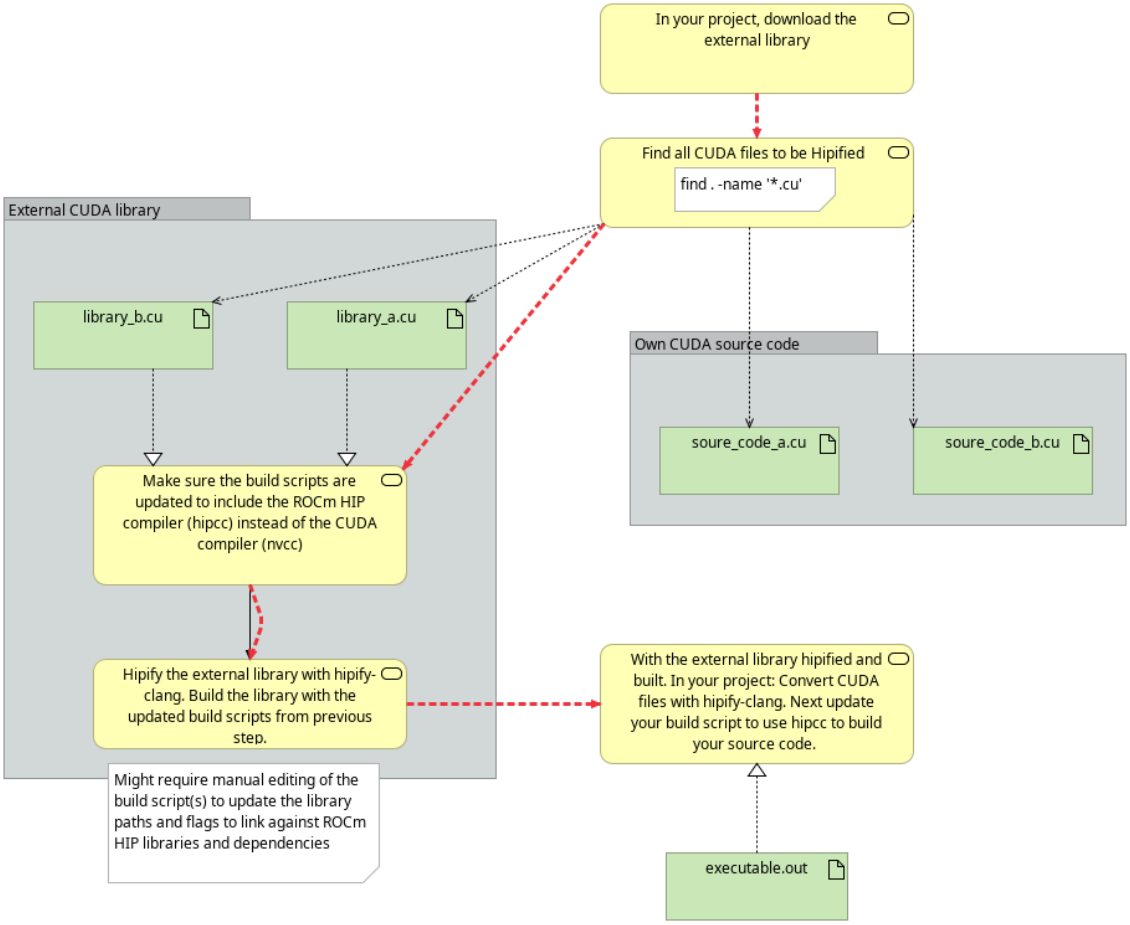

Figure 3.1 External library hipification with own code.

In the proposed general method for hipifying external CUDA libraries there are five steps highligted in yellow in the flowchart 3.1. The library is hipified with *hipify-clang*. In this general method the external library code build scripts are updated to use the *hipcc* compiler. After the library has been hipified and built successfully our own source code or target is hipified and compiled.

Chapter 4

Test Case

The test data that the QMR solver is run with is from DREAM (Disruption and Runaway Electron Analysis Model). DREAM is a modeling of instable electrons in a fusion reactor of the Tokamak type [24]. Under certain conditions, cascades of runaway electrons are created, which disrupt the properties of fusion plasmas and can lead to the dissolution of the plasma. The test data exists in the form of two matrices, Disruptions_A and Disriptions_b.

- **Disruptions_A**: 3 columns and 954401 rows (954401×3)
- **Disruptions_b**: 1 column and 87018 rows (87018×1)

The matrix Disruptions_A is given in the sparse matrix format: row, column, and value. The largest row value for the matrix Disruptions_A is 87017, and the largest column value is also 87017, meaning the A matrix is a quadratic sparse matrix of size 87017×87017 where rows and columns start at 0. The advantage of using QMR compared to the methods used before in DREAM is that QMR does not calculate A's inverse (which would be dense), and therefore we do not need to save a dense matrix in QMR. Additionally, we are using quadruple-precision (128-bit floating point) instead of double precision (64-bit).

Properties of the Disruptions_A matrix:

Number of rows:	87018
Number of elements:	954401
Number of zeroes:	93462
Number of nonzeros:	860939
Largest absolute value:	$A(40240, 86856) = +1.067052071992823800 \times 10^{26}$
Smallest absolute value:	$A(86744, 86759) = +5.305103115116799000 \times 10^{-75}$

Chapter 5

Results

The results are considered good if a program, depending on an external library, compiled with HIP, has roughly the same run times on hardware from both manufacturers, such as Nvidia and AMD. This is indeed the case with our QMR HIP implementation: the runtimes for consumer-grade AMD cards are comparable to those of an Professional-grade Nvidia Tesla V100-16GB, as illustrated in Figure 5.1 below.

Professional-grade GPUs typically offer higher precision and accuracy not only in floating-point arithmetic but also in other aspects such as double-precision floating-point arithmetic, error correction capabilities, larger memory capacities, optimized drivers for professional applications, and specialized features for compute-intensive workloads like scientific simulations, machine learning, and rendering.

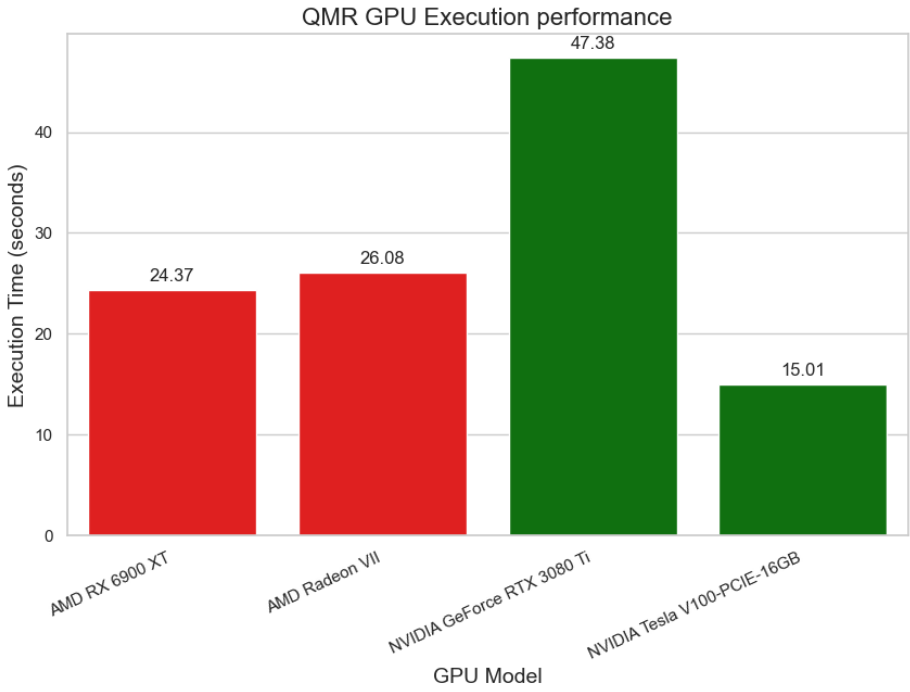

Figure 5.1 GPU performance comparison.

The results of the implementation are that it is now possible to hipify and compile CUDA programs to run on either manufacturer's hardware with external libraries that use CUDA. This opens up the possibility for developers and researchers to run older CUDA code on newer hardware by following the same process as outlined in this thesis. A process for hipifying CUDA code in libraries included within the current project has been presented in the form of a makefile, please see Appendix [7]. The porting or hipification will vary from project to project. The files needed to be converted from CUDA to HIP vary based on the project and the libraries used. This will need to be adjusted on a project-by-project basis, as shown in the flowchart figure 3.1.

Chapter 6

Discussion

The QMR implementation, which relied on SLEEF, did not require hipification to compile and run successfully; this was the expected easy path. The more challenging path, as presented in the methods chapter, involves hipifying the library code before compiling our own source code. The general method discussed in this thesis could potentially apply to other CUDA source code, as outlined in Figure 3.1. However, due to the method's generality, some manual porting from CUDA to HIP-compliant source code will be necessary. It's important to note that the tool hipify-perl functions as a simple "find and replace" script; for hipifying a codebase, hipify-clang should be utilized instead. In the future, automating the conversion of external libraries may be viable.

Chapter 7

Summary

This chapter will serve as a summary about the topics covered in this thesis.

There is a need to convert and run old CUDA programs hardware-agnostically. However, the process is not straightforward, and as of today, there does not exist any standard method for converting old CUDA programs that rely on external CUDA libraries to HIP.

In this master's thesis a general method is presented for converting existing CUDA programs that use an external library into HIP programs that can run on either hardware (Nvidia or AMD). AMD has developed a compiler along with conversion scripts that can take CUDA code and convert it into HIP code automatically. The problem that this thesis solves is that it presents a general method for converting existing CUDA programs that depend on a external library into HIP code that can target either manufacturers hardware.

Manual conversion work will be needed in most cases in order to satisfy the compiler (clang or NVCC through hipcc).

Chapter 8

Summary in Swedish - Svensk sammanfattning

Konvertering av CUDA program för att köra på AMD GPU:n

Introduktion

Superdatorer är idag mer i användning än någonsin, särskilt Graphics Processing Units (GPU:n) alltså grafikkorten i dessa datorer. Till exempel kan artificiell intelligens (AI) tränas på GPU:n, och andra parallelliserbara algoritmer kan också köras på GPU:n. Parallellprogrammering utgör ett programmeringsparadigm där ett givet program körs samtidigt på flera centralprocessorer (CPU:er) eller flera GPU:er. Den främsta fördelen med parallellprogrammering är den avsevärda ökningen av beräkningshastigheten.

Denna avhandling utforskar en lucka i den nuvarande forskningen om att kombinera tredjeparts-CUDA-bibliotek med befintlig källkod. Behovet av kvadrupelprecision är inte generellt, men i vårt Quasi-Minimal Residual method (QMR) problem med en miljoner variabler så får vi inte QMR att konvergera med en tolerans 10^{-22}. En float128 i C med kvadrupelprecision upptar 16 bytes (128 bitar) minne and har 33 signifikanta decimaler.

Syfte och motivering

Det primära syftet med denna avhandling är att möjliggöra för CUDA-kodutvecklare att köra sina program med externa bibliotek på antingen AMD- eller Nvidia-hårdvara.

Metod och Test Case

I denna avhandling presenteras en generell metod för att konvertera existerande CUDA källkod som använder sig av ett externt CUDA bibliotek till HIP kod som går att köra på både Nvidia och AMD grafikkort i superdatorer. I denna metod ingår *hipifiering* som är namnet på AMD:s process för att konvertera CUDA kod till HIP kod. Som test case kör vi ett QMR program som har input data: elektron disruptioner i fusionsreaktorer av Tokamak typen. Denna data ges i glesmatrisformatet: rad, kolumn, och värde.

Testdata som QMR-lösaren körs med kommer från DREAM (Disruption and Runaway Electron Analysis Model). DREAM är en modell av instabila elektroner i en fusionsreaktor av typen Tokamak [24]. Under vissa förhållanden skapas kaskader av elektroner som stör fusionsplasmats egenskaper och kan leda till att plasmat upplöses (inte hålls ihop). Testdatan finns i form av två matriser, Disruptions_A och Disruptions_b.

- **Disruptions_A**: 3 kolumner och 954401 rader (954401×3)
- **Disruptions_b**: 1 kolumn och 87018 rader (87018×1)

Disruptions_A matrisens egenskaper:

Antal rader:	87018
Antal element:	954401
Antal nollor:	93462
Antal icke-nollor:	860939
Största absoluta värdet:	$A(40240, 86856) = +1.067052071992823800 \times 10^{26}$
Minsta absoluta värdet:	$A(86744, 86759) = +5.305103115116799000 \times 10^{-75}$

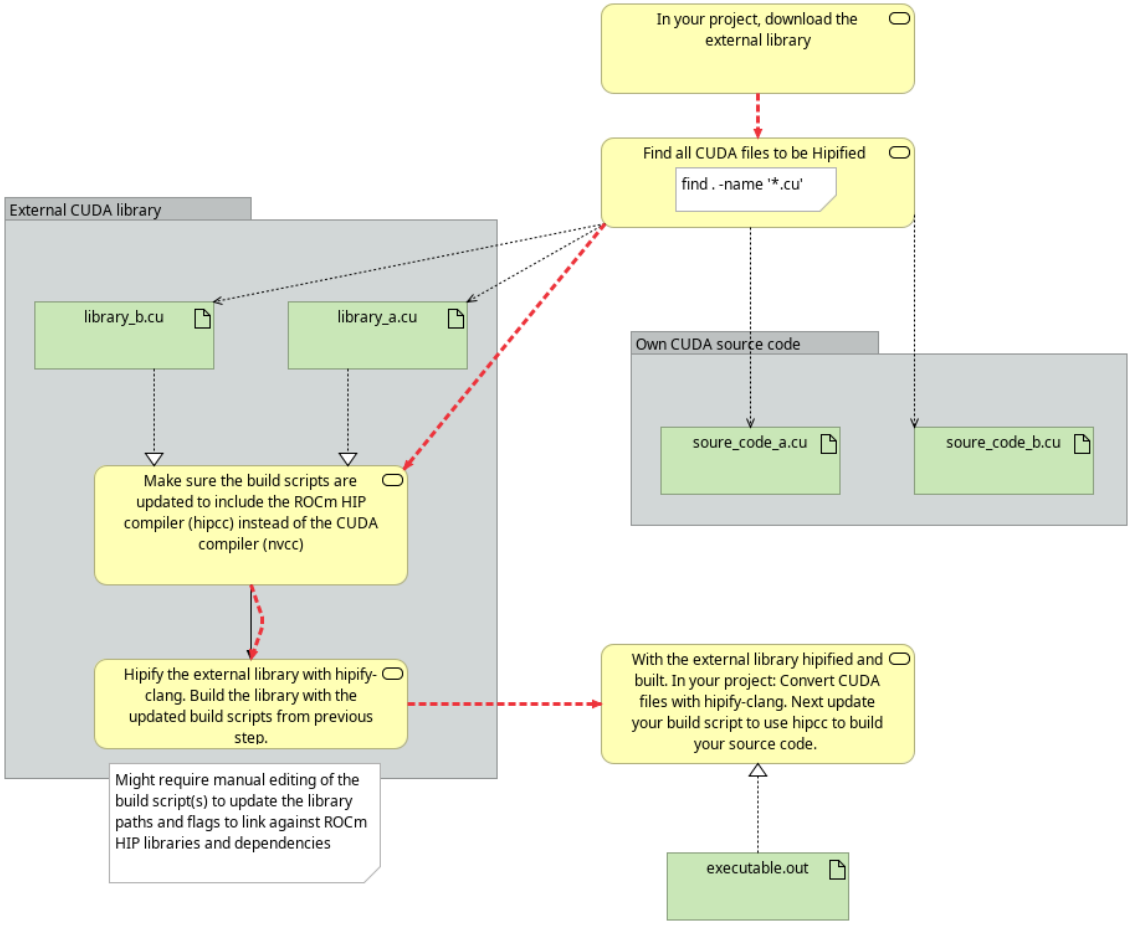

Figure 8.1 Generell metod för hipifiering av externt bibliotek tillsammans med egen källkod.

Den generella metoden 8.1 som togs fram i denna avhandling består av fem steg:

1. Ladda ner det externa biblioteket till ditt projekt.
2. Hitta alla CUDA filer med hjälp av Unix programmet *find*.
3. Gå igenom alla Makefiler och byggskript och byt ut kompilatorn till ROCm:s HIP kompilator som är: *hipcc*.
4. Hipifiera CUDA filer i det externa biblioteket med hjälp av: *hipify-clang*. Bygg sedan det externa biblioteket med de updatterade byggskripten från föregående steg.
5. Till sist så skall den egna koden hipifieras med *hipify-clang* och kompileras *hipcc*.

Resultat

Resultaten anses vara goda om ett program, beroende av ett externt bibliotek, kompilerat med HIP, har ungefär samma körtider på hårdvara från båda tillverkarna, såsom Nvidia och AMD. Och så är fallet med vårt QMR program som har körts på både AMD och Nvidia hårdvara med liknande körtider.

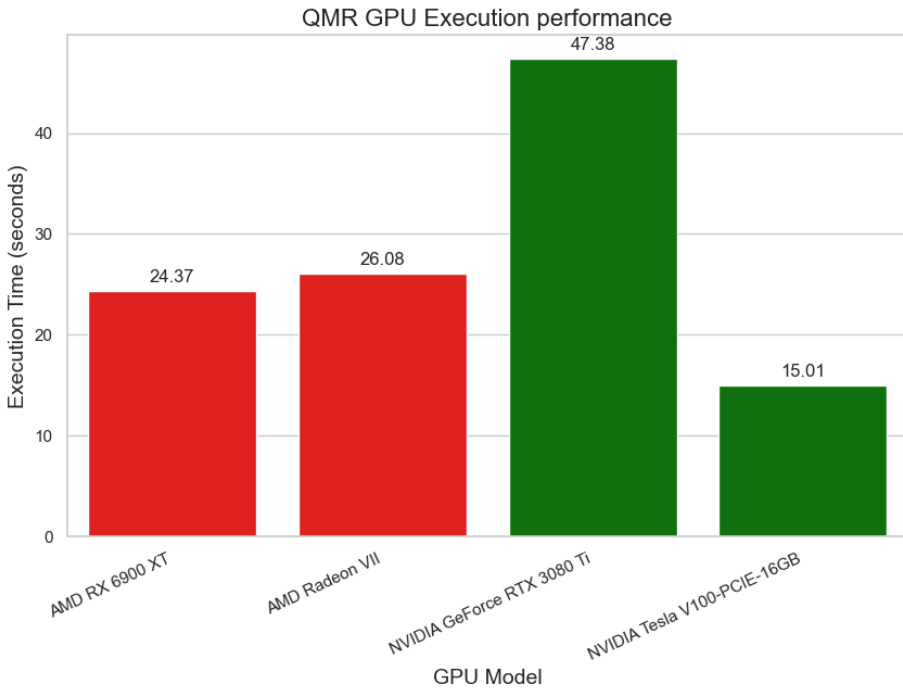

Figure 8.2 GPU prestanda jämförelse.

Att körtiderna avviker något från varandra beror på olika hårdvaruarkitekturer samt mängden dubbelprecisionsenheter i GPU:n. Spel-GPU:er, som till exempel AMD RX 6900 XT och Nvidia GeForce RTX 3080 Ti, fokuserar på många enkelprecisionberäkningsenheter för optimerad grafikrendering, medan vetenskapliga GPU:er, som Tesla V100, prioriterar ett större antal dubbelprecisionberäkningsenheter som är avgörande för vetenskapliga beräkningar och högpresterande databehandlingsuppgifter.

Slutsats

QMR-implementeringen, som använde sig av kvadrupelprecisionsbiblioteket SLEEF, krävde inte hipifiering för att kunna kompileras och köras framgångsrikt; detta var den förväntade enkla vägen. Den mer utmanande vägen, som presenterades i metodkapitlet, innebär att hipifiera den externa bibliotekskoden innan vi kompilerar vår egen källkod. Den generella metod som diskuteras i denna avhandling kan potentiellt tillämpas på annan CUDA-källkod, som beskrivs i Figur 8.1 Dock kommer på grund av metodens allmänhet, manuell konvertering från CUDA till HIP-kompatibel källkod att vara nödvändig. I framtiden kan det kanske vara möjligt att automatisera konverteringsprocessen för externa bibliotek. Så att man inte måste gå in manuellt i byggskripten för att byta ut kompilatorn och rätta till alla fel som kan uppstå i.o.m. bytet av kompilator.

References

[1] Intel, "Intel® arc™ a310 graphics," (2023), [Online; accessed 26-September-2023], URL https://ark.intel.com/content/www/us/en/ark/products/227958/intel-arc-a310-graphics.html.

[2] J. M. B. David H. Bailey, "High-precision computation and mathematical-physics," (2009 July 6), [Online; accessed 04-October-2023], URL https://www.davidhbailey.com/dhbpapers/dhb-jmb-acat08.pdf.

[3] AMD, "Amd releases new version of rocm, the most versatile open source platform for gpu computing," (2016, November 14), [Online; accessed 26-September-2023], URL https://ir.amd.com/news-events/press-releases/detail/732/amd-releases-new-version-of-rocm-the-most-versatile-open.

[4] L. H. S. a. C. I. C. f. S. L. George Markomanolis and F. R. T. S. at CSC IT Center for Science Ltd., "Preparing codes for lumi: converting cuda applications to hip," (15 April 2021), [Online; accessed 30-January-2024], URL https://www.lumi-supercomputer.eu/preparing-codes-for-lumi-converting-cuda-applications-to-hip/.

[5] E. H.-P. C. J. Undertaking, "Discover eurohpc ju," (2024), [Online; accessed 15-March-2024], URL https://eurohpc-ju.europa.eu/about/discover-eurohpc-ju_en.

[6] L. consortium, "Lumi consortium," (2024 March 13), [Online; accessed 15-March-2024], URL https://www.lumi-supercomputer.eu/lumi-consortium/.

[7] TOP500, "Top10 system - november 2023," (2023), [Online; accessed 07-December-2023], URL https://www.top500.org/lists/top500/2023/11/.

[8] LUMI, "Lumi's full system architecture revealed," (2021), [Online; accessed 26-September-2023], URL https://www.lumi-supercomputer.eu/lumis-full-system-architecture-revealed/.

[9] LUMI, "Gpu nodes - lumi-g," (2023), [Online; accessed 26-September-2023], URL https://docs.lumi-supercomputer.eu/hardware/lumig/.

[10] AMD, "Introduction to hip-faq," (Sep 29, 2022), [Online; accessed 07-November-2022], URL https://docs.amd.com/bundle/HIP-FAQ/page/Introduction_to_HIP-FAQ.html.

[11] AMD, "Programming with hip," (Oct 06, 2022), [Online; accessed 28-September-2023], URL https://docs.amd.com/projects/HIP/en/docs-5.1.0/user_guide/programming_manual.html#fma-and-contractions.

[12] J. Westerholm, "Gpu programming - slide set 2: Gpu programming model," (Oct 19, 2022), [Online; accessed 27-March-2023], URL https://moodle.abo.fi/pluginfile.php/164549/mod_resource/content/9/GPUProgramming2023_02.pdf.

[13] NVIDIA, "Cuda memory hierarchy," (27 February 2023), [Online; accessed 28-March-2023], URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-hierarchy.

[14] NVIDIA, "About cuda," (2023), [Online; accessed 25-July-2023], URL https://developer.nvidia.com/about-cuda.

[15] AMD, "What is rocm?," (2023 July 21), [Online; accessed 26-September-2023], URL https://rocm.docs.amd.com/en/latest/rocm.html.

[16] AMD, "Porting cuda vector add to hip," (2020), [Online; accessed 28-March-2023], URL https://www.amd.com/system/files/documents/chapter-4.1-porting-cuda-vector-add-to-hip.pdf.

[17] N. Shibata, "Sleef vectorized math library," (2021), [Online; accessed 30-January-2023], URL https://sleef.org/.

[18] F. P. Naoki Shibata, "Sleef: A portable vectorized library of c standard mathematical functions," (2020 June), [Online; accessed 28-September-2023], URL https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8936472.

[19] GNU, "quadmath_snprintf — convert to string," (27 July 2023), [Online; accessed 08-January-2024], URL https://gcc.gnu.org/onlinedocs/libquadmath/quadmath_005fsnprintf.html.

[20] Y. Saad, *Iterative Methods for Sparse Linear Systems* (SIAM, 2003), [Online; accessed 18-January-2024], URL https://www-users.cse.umn.edu/~saad/IterMethBook_2ndEd.pdf.

[21] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," SIAM Journal on Scientific and Statistical Computing **7**, 856 (1986), https://doi.org/10.1137/0907058.

[22] J. R. Shewchuk, Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University (1994), [Online; accessed 18-January-2024], URL https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

[23] D. Mukunoki and D. Takahashi, in *Parallel Processing and Applied Mathematics*, edited by R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski (Springer Berlin Heidelberg, Berlin, Heidelberg, 2014), pp. 632–642, ISBN 978-3-642-55224-3.

[24] M. Hoppe, O. Embreus, and T. Fülöp, "Dream: A fluid-kinetic framework for tokamak disruption runaway electron simulations," Computer Physics Communications **268**, 108098 (2021).

```
#include "hip/hip_runtime.h"
// Inspiration and code snippets borrowed from my lecturer Doctor Jan Westerholm at AAU.
// Victor Anderssen 2022 Fall

#include <hip/hip_runtime.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <inttypes.h>

long int MemoryAllocatedCPU = 0L;

#define gpuErrchk(ans)                    \
  {                                       \
    gpuAssert((ans), __FILE__, __LINE__); \
  }
inline void gpuAssert(hipError_t code, const char *file, int line, bool abort = true)
{
  if (code != hipSuccess)
  {
    fprintf(stderr, "GPUassert: %s %s %d\n", hipGetErrorString(code), file, line);
    if (abort)
      exit(code);
  }
}

/* @Note(Victor):
  * All thread/kernel receives these four params as the dim3 type
  *
  * gridDim : gridDim.x, gridDim.y, gridDim.z
  * blockIdx : blockIdx.x, blockIdx.y, blockIdx.z
  * blockDim : blockDim.x, blockDim.y, blockDim.z
  * threadIdx: threadIdx.x, threadIdx.y, threadIdx.z */
__global__ void measure_kernel_memory_transfer_overhead_kernel()
{
}

static int
get_device()
{
  int deviceCount;
  hipGetDeviceCount(&deviceCount);
```

```
44  printf("  Found %d CUDA devices\n", deviceCount);
45
46  if (deviceCount < 0 || deviceCount > 128)
47  {
48    return (-1);
49  }
50
51  int device;
52  for (device = 0; device < deviceCount; ++device)
53  {
54    hipDeviceProp_t deviceProp;
55    hipGetDeviceProperties(&deviceProp, device);
56    printf("\tDevice %s                 = device %d\n", deviceProp.name, device);
57    printf("\tcompute capability       =        %d.%d\n", deviceProp.major,
          deviceProp.minor);
58    printf("\ttotalGlobalMemory        =       %.2lf GB\n",
          deviceProp.totalGlobalMem / 1000000000.0);
59    printf("\tl2CacheSize              =    %8d B\n", deviceProp.l2CacheSize);
60    printf("\tregsPerBlock             =    %8d\n", deviceProp.regsPerBlock);
61    printf("\tmultiProcessorCount      =    %8d\n",
          deviceProp.multiProcessorCount);
62    printf("\tmaxThreadsPerMultiprocessor = %8d\n",
          deviceProp.maxThreadsPerMultiProcessor);
63    printf("\tsharedMemPerBlock        =    %8d B\n",
          (int)deviceProp.sharedMemPerBlock);
64    printf("\twarpSize                 =    %8d\n", deviceProp.warpSize);
65    printf("\tclockRate                =    %8.2lf MHz\n", deviceProp.clockRate
          / 1000.0);
66    printf("\tmaxThreadsPerBlock       =    %8d\n",
          deviceProp.maxThreadsPerBlock);
67    printf("\tmaxGridSize              =    %d x %d x %d\n",
68        deviceProp.maxGridSize[0], deviceProp.maxGridSize[1],
              deviceProp.maxGridSize[2]);
69    printf("\tmaxThreadsDim            =    %d x %d x %d\n",
70        deviceProp.maxThreadsDim[0], deviceProp.maxThreadsDim[1],
              deviceProp.maxThreadsDim[2]);
71    printf("\tconcurrentKernels        =    ");
72  }
73
74  hipSetDevice(0);
75  hipGetDevice(&device);
76
77  if (device != 0)
78  {
79    printf("  Unable to set device 0, using %d instead", device);
80  }
81  else
```

```
82    {
83      printf("  Using CUDA device %d\n\n", device);
84    }
85
86    return (0);
87  }
88
89  int main(void)
90  {
91    printf("       Starting the program\n");
92    get_device();
93
94    struct timeval st, et;
95    struct timezone _tzone;
96    const unsigned long N = 1L;
97    gettimeofday(&st, &_tzone);
98
99    dim3 threadsInBlock(1, 1);
100   dim3 blocksInGrid = dim3(ceil((N + threadsInBlock.x - 1) / threadsInBlock.x),
101             ceil((N + threadsInBlock.y - 1) / threadsInBlock.y));
102
103   printf("======================================================================$\$n");
104   printf("blocksInGrid:\t{%d, %d, %d} blocks.\nthreadsInBlock:\t%d threads.\n",
105       blocksInGrid.x, blocksInGrid.y, blocksInGrid.z, threadsInBlock.x *
              threadsInBlock.y * threadsInBlock.z);
106
107   const long int number_of_threads = (long int)(threadsInBlock.x *
          ((long)(threadsInBlock.y)) * threadsInBlock.z * ((blocksInGrid.x *
          blocksInGrid.y) * blocksInGrid.z));
108
109   printf("number of threads: %ld\n", number_of_threads);
110
111   // Call the GPU kernel(s)
112   hipLaunchKernelGGL(measure_kernel_memory_transfer_overhead_kernel,
          blocksInGrid, threadsInBlock, 0, 0);
113
114   gpuErrchk(hipGetLastError());
115   gpuErrchk(hipDeviceSynchronize());
116
117   printf("       Total memory allocated = %.1lf MB\n", MemoryAllocatedCPU /
          1000000.0);
118   gettimeofday(&et, &_tzone);
119
120   int elapsed = ((et.tv_sec - st.tv_sec) * 1000000) + (et.tv_usec - st.tv_usec);
121   printf("       The program took %d microseconds\n", elapsed);
122   printf("       The program took %d milliseconds\n", elapsed / 1000);
```

```
  printf("       The program took %f seconds\n", double((double)elapsed /
      1000000.0));
  printf("       To execute the GPU kernel\n");

  return (0);
}
```

Appendices 2 Loop independency in matrix multiplication in C and CUDA.

```
#include <stdio.h>
#include <cuda_runtime.h>

// CPU version
void matrixMultiply(float* A, float* B, float* C, int m, int n, int k) {
    for (int row = 0; row < m; ++row) {
        for (int col = 0; col < k; ++col) {
            float sum = 0.0f;
            for (int i = 0; i < n; ++i) {
                sum += A[row * n + i] * B[i * k + col];
            }
            C[row * k + col] = sum;
        }
    }
}

// CUDA (GPU) version
__global__ void matrixMultiply(float* A, float* B, float* C, int m, int n, int
     k) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m && col < k) {
        float sum = 0.0f;
        for (int i = 0; i < n; ++i) {
            sum += A[row * n + i] * B[i * k + col];
        }
        C[row * k + col] = sum;
    }
}

int main() {
    // Host code
    // ...

    // Function call
    matrixMultiply(h_A, h_B, h_C, m, n, k);

```

```
    // CUDA kernel launch
    matrixMultiply<<<grid_size, block_size>>>(d_A, d_B, d_C, m, n, k);

    // Copy results back to host
    // ...

    // Cleanup
    // ...

    return 0;
}
```

Appendices 3 How to compile the Example HIP program

```
# Compile
hipcc main.cpp -o example_hip_program

# Run
./example_hip_program
```

Appendices 4 Example program output

```
$ make

Cleaning
rm -r build 2> /dev/null || true
rm -r code/*.cu 2> /dev/null || true

Creating directories
mkdir -p build

Hipifying the Cuda C++ code to HIP C++ code
hipify-perl ./code/main.cpp -o ./code/main.cpp.hip.cu

Building the program
hipcc -O3 ./code/main.cpp.hip.cu -o ./build/gpu_signal_processing.out

Running the executable
./build/gpu_signal_processing.out
       Starting the program
  Found 1 CUDA devices
        Device AMD Radeon RX 6900 XT               = device 0
        compute capability         =        10.3
        totalGlobalMemory          =       17.16 GB
        l2CacheSize                =     4194304 B
        regsPerBlock               =       65536
```

```
        multiProcessorCount       =         40
        maxThreadsPerMultiprocessor =      2048
        sharedMemPerBlock         =      65536 B
        warpSize                  =         32
        clockRate                 =    2660.00 MHz
        maxThreadsPerBlock        =       1024
        maxGridSize               =    2147483647 x 2147483647 x 2147483647
        maxThreadsDim             =    1024 x 1024 x 1024
   Using CUDA device 0

========================================================================
blocksInGrid: {1, 1, 1} blocks.
threadsInBlock: 1024 threads.
number of threads: 1024
       The program took 156325 microseconds
       The program took 156 milliseconds
       The program took 0.156325 seconds
       To execute the GPU kernel

The program has been built and run successfully!
```

Appendices 5 Download SLEEF from github shell script

```
#!/usr/bin/bash
if [ ! -d "./sleef" ]
then
  # Clone if not existing
  echo "Cloning SLEEF"
  git clone https://github.com/shibatch/sleef.git
fi
```

Appendices 6 Install SLEEF locally shell script

```
#!/usr/bin/bash
cd sleef

mkdir -p build
cd build
pwd

cmake .. -L

cmake -DBUILD_INLINE_HEADERS=TRUE \
    -DBUILD_QUAD=TRUE \
    -DENABLE_CUDA=TRUE \
    -DCMAKE_CUDA_ARCHITECTURE=86 \
```

```
    -DCMAKE_BUILD_TYPE=Debug \
    ..

make
# make test
cd ../../
```

Appendices 7 A Build makefile

```
# print arbitrary variables with $ make print-<name>
print-% : ; @echo $* = $($*)

# define the C++ compiler to use
CC = hipcc # which is actually clang

# define any compile-time flags
CXXFLAGS  := -O3 -march=native -ffp-contract=off # the flag -ffp-contract=off is
    crucial !!!
CXXFLAGS  += -I/usr/lib/gcc/x86_64-pc-linux-gnu/12.2.1/include
CXXFLAGS  += -I./sleef/build/include
CXXFLAGS  += -I/opt/rocm/include

# Linker flags
LDFLAGS   := -lm -fopenmp -lquadmath

# Library includes
LDLIBS    := # -L/usr/lib/aomp_15.0-3/lib

# Define the source and destination
SRC_DIR   := ./code
OBJ_DIR   := ./build
SRC_FILES := $(shell find ./code -name '*.cu')

# define the executable file, the program name
TARGET = a

.PHONY: clean

# all: clean dirs download_sleef hipify_sleef install_sleef hipify $(TARGET)
all: clean dirs download_sleef install_sleef hipify $(TARGET)
  @echo
  @echo        The program has been built run successfully!

hipify_sleef:
  @echo
  @echo    Hipifying Sleef
  $(shell export HIP_PLATFORM=amd)
```

```
38    $(shell export PLATFORM=amd)
39    $(shell export LD_LIBRARY_PATH=/usr/lib/aomp_15.0-3/lib)
40
41    # Take backups of the CUDA source files
42    mv ./sleef/src/quad-tester/qiutcuda.cu
          ./sleef/src/quad-tester/qiutcuda.cu.original
43    mv ./sleef/src/libm-tester/iutcuda.cu
          ./sleef/src/libm-tester/iutcuda.cu.original
44
45    # Hipify the CUDA source files, passing the compiler flags to hipcc.
46    hipify-clang -v ./sleef/src/quad-tester/qiutcuda.cu.original
          -o=./sleef/src/quad-tester/qiutcuda.cu -- -Xclang $(CXXFLAGS) $(LDFLAGS)
          $(LDLIBS)
47    hipify-clang -v ./sleef/src/libm-tester/iutcuda.cu.original
          -o=./sleef/src/libm-tester/iutcuda.cu -- -Xclang $(CXXFLAGS) $(LDFLAGS)
          $(LDLIBS)
48
49  download_sleef:
50    @echo
51    @echo  Downloading Sleef...
52    ./download_sleef_from_github
53
54  install_sleef:
55    @echo
56    @echo  Installing Sleef locally
57    ./install_sleef_locally
58
59  hipify:
60    @echo
61    @echo        Hipifying the Cuda C++ code to HIP C++ code
62
63    # Slow method
64    # hipify-perl ./code/main.cu -o ./build/main.hip.cpp
65    # hipify-perl ./code/qmr.cu -o ./build/qmr.hip.cpp
66
67    # Fast method
68    hipify-clang ./code/main.cu -o=build/main.hip.cpp -v --print-stats -- -Xclang
          $(CXXFLAGS) $(LDFLAGS) $(LDLIBS)
69    hipify-clang ./code/qmr.cu -o=build/qmr.hip.cpp -v --print-stats -- -Xclang
          $(CXXFLAGS) $(LDFLAGS) $(LDLIBS)
70
71  $(TARGET):
72    @echo
73    @echo       Building the program with hipcc
74    $(CC) $(CXXFLAGS) $(LDFLAGS) $(LDLIBS) ./build/main.hip.cpp ./build/qmr.hip.cpp
75
76    @echo
```

```
77    @echo       Running the executable
78    ./run_qmr
79
80  dirs:
81    @echo
82    @echo       Creating directories
83    mkdir -p build
84
85  clean:
86    @echo
87    @echo       Cleaning
88    rm -r build 2> /dev/null || true
89    rm a.out 2> /dev/null || true
90
91  run:
92    @echo
93    @echo       Running the executable
94    ./build/$(TARGET).out
```

Appendices 8 Implementation of IEE754 quadruple precision with bit manipulation

```
1   #include <stdio.h>
2   #include <stdint.h>
3
4   // Define masks for extracting components
5   #define SIGN_MASK 0x8000000000000000ULL
6   #define EXPONENT_MASK 0x7FF0000000000000ULL
7   #define SIGNIFICAND_MASK 0x000FFFFFFFFFFFFFULL
8   #define EXPONENT_BIAS 1023
9
10  typedef struct
11  {
12    uint64_t sign;
13    uint64_t exponent;
14    uint64_t significand;
15  } QuadruplePrecision;
16
17  // Convert a double-precision floating-point number to quadruple precision.
18  void doubleToQuadruple(double d, QuadruplePrecision *q)
19  {
20    // Extract the bit representation of the double-precision number.
21    uint64_t doubleBits = *((uint64_t *)&d);
22
23    // Extract and store the sign bit of the quadruple-precision representation.
24    q->sign = (doubleBits & SIGN_MASK);
25
```

```
26    // Extract and store the exponent bits from the double-precision number and
          adjust them for quadruple precision.
27    q->exponent = (doubleBits & EXPONENT_MASK) >> 52;
28    if (q->exponent == 0)
29    {
30      // Handle denormalized numbers by setting the quadruple-precision exponent
            to 1.
31      q->exponent = 1;
32    }
33    else
34    {
35      // Adjust the exponent bias for quadruple precision.
36      q->exponent = q->exponent - EXPONENT_BIAS + 16383;
37    }
38
39    // Extract and store the significand bits and shift them to align with
          quadruple precision.
40    q->significand = (doubleBits & SIGNIFICAND_MASK) << 11;
41  }
42
43  // Convert a quadruple-precision floating-point representation to a
        double-precision number.
44  double quadrupleToDouble(QuadruplePrecision *q)
45  {
46    // Combine the sign bit, adjusted exponent, and significand to reconstruct the
          bit pattern of a double-precision number.
47    uint64_t doubleBits = q->sign | ((q->exponent - 16383 + EXPONENT_BIAS) << 52)
          | (q->significand >> 11);
48
49    // Interpret the bit pattern as a double-precision number and return it.
50    return *((double *)&doubleBits);
51  }
52
53  int main(void)
54  {
55    // Two double-precision numbers
56    double double1 = 1.337;
57    double double2 = 4.200;
58
59    // Declare quadruple-precision variables to store the converted values
60    QuadruplePrecision quadruple1, quadruple2;
61
62    // Convert the double-precision numbers to quadruple-precision
63    doubleToQuadruple(double1, &quadruple1);
64    doubleToQuadruple(double2, &quadruple2);
65
66    // Convert the quadruple-precision numbers back to double-precision
```

```
  double result1 = quadrupleToDouble(&quadruple1);
  double result2 = quadrupleToDouble(&quadruple2);

  // Add the double-precision numbers in quadruple-precision format
  long double result = result1 + result2;

  // Print the results with high precision
  printf("Double 1: \t\t\t%.40f\n", result1);
  printf("Double 2: \t\t\t%.40f\n", result2);
  printf("Quadruple-precision result:\t%.40Lf\n", result);

  return 0;
}
```

Appendices 9 SLEEF implementation addition with high precision numbers

```
#include <quadmath.h>
#include <stdio.h>
#include <sleef.h>
#include <sleefquad.h>

int main(int argc, char **argv)
{
    printf("\n\tSleef Add example\n");

    // Define two __float128 numbers
    Sleef_quad NumberA = 1.337Q;
    Sleef_quad NumberB = 4.200Q;

    // Add the two __float128 numbers together
    Sleef_quad Sum = Sleef_addq1_u05(NumberA, NumberB);

    // Print the result
    Sleef_printf("\tNumberA: %.40Pg\n", &NumberA);
    Sleef_printf("\tNumberB: %.40Pg\n", &NumberB);
    Sleef_printf("\tSum:\t %.40Pg\n", &Sum);

    return (0);
}
```

Appendices 10 SLEEF implementation multiplication with high precision numbers

```
#include <quadmath.h>
#include <stdio.h>
#include <sleefquad.h>

```

```
int main(int argc, char **argv)
{
    printf("\n\tSleef Multiply example\n");

    // Define two __float128 numbers
    Sleef_quad NumberA = 1.337Q;
    Sleef_quad NumberB = 4.200Q;

    // Multiply the two __float128 numbers together
    Sleef_quad Sum = Sleef_mulq1_u05(NumberA, NumberB);

    // Print the result
    Sleef_printf("\tNumberA: %.40Pg\n", &NumberA);
    Sleef_printf("\tNumberB: %.40Pg\n", &NumberB);
    Sleef_printf("\tSum:\t %.40Pg\n", &Sum);

    return (0);
}
```

Appendices 11 SLEEF implementation of printing high precision numbers with quadmath

```
#include <stdio.h>
#include <quadmath.h>

int main(void)
{
    // Define a known _Float128 value
    _Float128 PI = acosq(-1.0Q);

    // Buffer to hold the formatted string
    char Buffer[100]; // Adjust the size accordingly

    // Format the _Float128 value to a string with specific precision
    int DecimalPrecision = 34;
    int Result = quadmath_snprintf(Buffer, sizeof(Buffer), "%.*Qf",
         DecimalPrecision, PI);

    // Check if the formatting was successful
    if (Result < 0)
    {
        printf("\n\t[ERROR]: Formatting failed!\n");
        return (-1);
    }

    // Display the result with 33 decimal places
```

```
24      printf("\n\tAsserted: acos(-1.0Q) with %d decimals of precision: %s\n\n",
            DecimalPrecision, Buffer);
25
26      return (0);
27  }
```

Appendices 12 Build script for all the SLEEF API examples

```
1   #!/usr/bin/bash
2
3   # If the dir sleef does not exist run the block
4   if [ ! -d "./sleef" ]
5   then
6       # Create build folder if not existing
7       mkdir -p build 2> /dev/null || true
8
9       # Run the download_sleef_from_github bash script
10      /bin/bash ./download_sleef_from_github.sh
11      /bin/bash ./install_sleef_locally.sh
12  fi
13
14  # Set the CFLAGS environment variable to include the Sleef headers.
15  # define any compile-time flags
16  export CXXFLAGS="-O3 -Wattributes -I./sleef/build/include
        -I/usr/lib/gcc/x86_64-pc-linux-gnu/13.2.1/include"
17  export LDLIBS="-L ./sleef/build/lib"
18  export LDFLAGS="-lquadmath -lsleef -lsleefquad -lm"
19  export LD_LIBRARY_PATH="./sleef/build/lib"
20
21  # Build the example with clang and link against the Sleef library.
22  gcc -o ./build/sleef_add_example $CXXFLAGS $LDLIBS $LDFLAGS sleef_add_example.c
23  gcc -o ./build/sleef_multiply_example $CXXFLAGS $LDLIBS $LDFLAGS
        sleef_multiply_example.c
24  gcc -o ./build/sleef_print_example $CXXFLAGS $LDLIBS $LDFLAGS
        sleef_print_example.c
25
26  # Run the examples
27  ./build/sleef_add_example
28  ./build/sleef_multiply_example
29  ./build/sleef_print_example
```

Appendices 13 HIP Vector addition example.

```
// HIP Vector Addition
#include <iostream>
#include <hip/hip_runtime.h>

__global__
void vectorAdd(int *a, int *b, int *c, int size) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size)
        c[i] = a[i] + b[i];
}

int main(void) {
    const int size = 1024;
    int a[size], b[size], c[size];

    // Initialize input vectors
    for (int i = 0; i < size; ++i) {
        a[i] = i;
        b[i] = 2 * i;
    }

    // Allocate device memory
    int *d_a, *d_b, *d_c;
    hipMalloc((void**)&d_a, size * sizeof(int));
    hipMalloc((void**)&d_b, size * sizeof(int));
    hipMalloc((void**)&d_c, size * sizeof(int));

    // Copy data from host to device
    hipMemcpy(d_a, a, size * sizeof(int), hipMemcpyHostToDevice);
    hipMemcpy(d_b, b, size * sizeof(int), hipMemcpyHostToDevice);

    hipLaunchKernelGGL(vectorAdd, dim3((size + 255) / 256), dim3(256), 0, 0,
         d_a, d_b, d_c, size);

    // Copy result from device to host
    hipMemcpy(c, d_c, size * sizeof(int), hipMemcpyDeviceToHost);

    hipFree(d_a); // Free device memory
    hipFree(d_b);
    hipFree(d_c);

    for (int i = 0; i < 10; ++i) // Print result
        std::cout << c[i] << " ";

    return (0);
}
```

Appendices 14 Linear equation example.

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
    // Coefficients of the linear system
    double CoefficientA11 = 2.0, CoefficientA12 = 3.0, ConstantB1 = 8.0;
    double CoefficientA21 = -1.0, CoefficientA22 = 2.0, ConstantB2 = 3.0;

    // Variables
    double SolutionX, SolutionY;

    // Solving the linear system
    SolutionX = (ConstantB1 * CoefficientA22 - ConstantB2 * CoefficientA12) /
         (CoefficientA11 * CoefficientA22 - CoefficientA12 * CoefficientA21);
    SolutionY = (ConstantB2 * CoefficientA11 - ConstantB1 * CoefficientA21) /
         (CoefficientA11 * CoefficientA22 - CoefficientA12 * CoefficientA21);

    // Display the solution
    printf("\tSolution:\n");
    printf("\tx = %.2f\n", SolutionX);
    printf("\ty = %.2f\n", SolutionY);

    // Using mathematical constants
    printf("\n\tSome constants:\n");
    printf("\tPI = %.6f\n", M_PI);
    printf("\tEuler's number (e) = %.6f\n", M_E);

    return (0);
}
```

Appendices 15 Shared memory example in HIP.

```c
#include <iostream>
#include <hip/hip_runtime.h>

const unsigned long int N = 1000000UL; // Vector size
const unsigned int SHARED_MEM_SIZE = 1024;

#define gpuErrchk(ans)                    \
    {                                     \
        gpuAssert((ans), __FILE__, __LINE__); \
    }
inline void gpuAssert(hipError_t code, const char *file, int line, bool abort =
     true)
{
```

```
13      if (code != hipSuccess)
14      {
15          fprintf(stderr, "GPUassert: %s %s %d\n", hipGetErrorString(code), file,
                line);
16          if (abort)
17              exit(code);
18      }
19  }
20
21  __global__ void vectorAdd(float *a, float *b, float *c)
22  {
23      int tid = threadIdx.x + blockIdx.x * blockDim.x;
24
25      __shared__ float shared_c[SHARED_MEM_SIZE]; // Shared memory for each block
26
27      shared_c[threadIdx.x] = 0.0; // Initialize shared memory to 0
28
29      while (tid < N) // Perform vector addition using shared memory
30      {
31          shared_c[threadIdx.x] += a[tid] + b[tid];
32          tid += blockDim.x * gridDim.x;
33      }
34
35      __syncthreads(); // Synchronize within the block before performing reduction
36
37      // Perform parallel reduction using shared memory
38      for (int stride = blockDim.x / 2; stride > 0; stride >>= 1)
39      {
40          if (threadIdx.x < stride)
41          {
42              shared_c[threadIdx.x] += shared_c[threadIdx.x + stride];
43          }
44          __syncthreads();
45      }
46
47      // Store the result to global memory for each block
48      if (threadIdx.x == 0)
49      {
50          c[blockIdx.x] = shared_c[0];
51      }
52  }
53
54  int main(void)
55  {
56      // Host vectors
57      float *h_a, *h_b, *h_c;
58
```

```
59      // Device vectors
60      float *d_a, *d_b, *d_c;
61
62      // Allocate memory on the host
63      h_a = new float[N];
64      h_b = new float[N];
65      h_c = new float[SHARED_MEM_SIZE]; // Allocate enough space for each block's
            result
66
67      // Initialize host vectors
68      for (int i = 0; i < N; ++i)
69      {
70          h_a[i] = i;
71          h_b[i] = i * 2;
72      }
73
74      // Allocate memory on the device
75      gpuErrchk(hipMalloc((void **)&d_a, N * sizeof(float)));
76      gpuErrchk(hipMalloc((void **)&d_b, N * sizeof(float)));
77      gpuErrchk(hipMalloc((void **)&d_c, SHARED_MEM_SIZE * sizeof(float))); //
            Each block has its own result
78
79      // Copy host data to device
80      gpuErrchk(hipMemcpy(d_a, h_a, N * sizeof(float), hipMemcpyHostToDevice));
81      gpuErrchk(hipMemcpy(d_b, h_b, N * sizeof(float), hipMemcpyHostToDevice));
82
83      // Configure the grid and block dimensions
84      const int BlockSize = 1024;
85      dim3 blockDim(BlockSize, 1, 1);
86
87      // Calculate the number of blocks needed
88      int NumBlocks = (N + blockDim.x - 1) / blockDim.x;
89
90      dim3 gridDim(NumBlocks, 1, 1);
91
92      // Print configuration
93      printf("\tGrid dimensions: %d x %d x %d\n", gridDim.x, gridDim.y, gridDim.z);
94      printf("\tBlock dimensions: %d x %d x %d\n", blockDim.x, blockDim.y,
            blockDim.z);
95      printf("\tNumber of threads: %d\n", gridDim.x * blockDim.x);
96
97      // Ensure all kernel launches are complete
98      gpuErrchk(hipDeviceSynchronize());
99
100     // Launch the kernel
101     vectorAdd<<<gridDim, blockDim, SHARED_MEM_SIZE * sizeof(float)>>>(d_a, d_b,
            d_c);
```

```
102
103     // Ensure all kernel launches are complete
104     gpuErrchk(hipDeviceSynchronize());
105
106     // Copy the result back to the host
107     gpuErrchk(hipMemcpy(h_c, d_c, NumBlocks * sizeof(float),
            hipMemcpyDeviceToHost));
108
109     // Ensure all kernel launches are complete
110     gpuErrchk(hipDeviceSynchronize());
111
112     // Perform reduction on the CPU side
113     float result = 0.0;
114     for (int i = 0; i < NumBlocks; ++i)
115     {
116         result += h_c[i];
117     }
118
119     // Verify the result
120     const unsigned long expected_result = N * (N - 1) / 2 * 3; // Sum of
            arithmetic sequence
121     if (std::fabs(result - (float)expected_result) > 1e-1 * expected_result)
122     {
123         std::cerr << "\t\t[ERROR]: Verification failed!" << std::endl;
124         std::cerr << "\t\t[ERROR]: Expected result: " << expected_result <<
                std::endl;
125         std::cerr << "\t\t[ERROR]: Actual result: " << result << std::endl;
126         std::cerr << "\t\t[ERROR]: Error:       " << std::fabs(result -
                (float)expected_result) << std::endl;
127
128         // Free allocated memory
129         delete[] h_a;
130         delete[] h_b;
131         delete[] h_c;
132
133         gpuErrchk(hipFree(d_a));
134         gpuErrchk(hipFree(d_b));
135         gpuErrchk(hipFree(d_c));
136
137         return (1);
138     }
139
140     printf("\tVector addition successful!\n");
141     std::cout << "\tExpected result: " << expected_result << std::endl;
142     std::cout << "\tActual result: " << result << std::endl;
143
144     // Free allocated memory
```

```
145     delete[] h_a;
146     delete[] h_b;
147     delete[] h_c;
148
149     gpuErrchk(hipFree(d_a));
150     gpuErrchk(hipFree(d_b));
151     gpuErrchk(hipFree(d_c));
152
153     return (0);
154 }
```