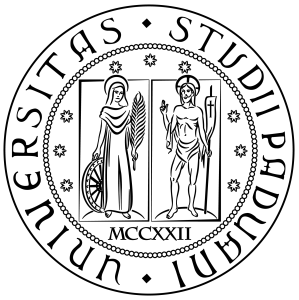

University of Padova
Department of Information Engineering
Master's Degree in Computer Engineering

High Performance Sparse Iterative Solvers for Heterogeneous Systems: Optimization and Linear Algebra

Supervisor:
Prof. Francesco Silvestri

Co-supervisor:
Dr. Denis Jelovina

Candidate:
Giovanni Gaio

Student ID:
2149760

---

academic year 2025-2026

Abstract

The thesis is divided into two parts: the first presents the implementation of two optimization algorithms based on the ALP/GraphBLAS library, the second part analyzes the theory and implementation of an Iterative Refinement (IR) technique for solving sparse linear systems at high precision on hardware only supporting low-precision floating point operations.

Simulated Annealing-Replica Exchange (SARE) and Simulated Bifurcation (SB) are two physics-inspired general optimization techniques. The first is naturally expressed in a Single Program Multiple Data fashion, which naturally maps to a multiprocess implementation. This easily combines with the multithreaded GraphBLAS library to form a hybrid implementation. SB is naturally expressed using GraphBLAS' linear algebra primitives.

Both implementations are evaluated on selected problems of the G-set collection. In a single-threaded setting, these solvers show competitive performance. Furthermore, SARE shows good scalability up to 4 nodes, while SB scales well, quickly reaching the memory bandwidth limit.

The novel IR technique shown in this work can be seen as a combination of classical IR methods with an Ozaki scheme. The matrix is decomposed into low-precision parts: all are used to keep a precise estimate of the residual as in a standard Ozaki scheme, while the update direction is computed using a half-precision Conjugate Gradient (CG).

We evaluate this IR method on a set of Symmetric Positive Definite sparse matrices with controlled condition numbers and densities. In this setting, the runtime of the IR is similar to some multithreaded CPU implementations of CG working in double precision, while reliably achieving double-precision accuracy on well-conditioned matrices.

Contents

Chapter 1

Introduction

The research presented in this work was done during an internship at Huawei's Computing Systems Laboratory in Zurich, under the supervision of Denis Jelovina.

In recent years the search of more computing performance has taken a turn towards more and more specific hardware. The challenges of miniaturization have meant that Moore's Law has slowed; at the same time Dennard Scaling has stopped, meaning that smaller transistors or higher frequencies are not viable options to increase chips' performance.

Therefore hardware specialization and parallelism are more important than ever to improve computational performance. Specialized hardware can make better use of a fixed amount of silicon, by mapping the problem structure into physical connections. Parallelism, on the other hand, is applied on many levels: instruction-level, within a core with pipelines, thread-level, between the cores of a Central Processing Unit (CPU), and process-level for multiple CPUs and multiple machines.

A good example of this trend is the scale and cost of Artificial Intelligence (AI) require the development of specialized hardware. But this trend is also apparent across the field of High Performance Computing (HPC), where heterogeneous systems are commonly used.

The guiding goal throughout the thesis is to develop high performance solvers for selected optimization problems of known significance. A search for high performance implies maximizing parallelism, attention to data movements, and the use of specialized hardware and software frameworks.

We explore multiple implementations of high performance solvers, but using very different techniques for different problems. First, we consider solvers for sparse optimization problems using proven sparse linear algebra primitives. Then we detail a novel linear systems solver that uses highly parallel hardware to accelerate linear algebra operations.

The developed algorithms share some interesting characteristics, that exemplify some noteworthy trends. We detail below some of these common features and their significance.

Iterative Many methods and algorithms are iterative in nature, since in general it is relatively easy to find a locally good direction, solve a small part of the given problem, or simply find an approximate solution.

In the two case studies of this thesis we explore two iterative methods. They are similar in the fact that each iteration solves the same problem using the same technique. At each iteration, the algorithms improve the current solution by local optimization.

But the nature of the iterative step is different: optimization algorithms iteratively improve a solution on the same space, scale and objective; in linear algebra, the idea is to refine the solution in a way that the exploration concentrates near the solution and at a smaller scale. In linear algebra such Iterative Refinement (IR) methods are common, as they are a simple way to achieve a solution accuracy higher than native hardware capabilities.

This difference is dictated by the nature of the solution space: in one case discrete and non-monotonic, in the other continuous, smooth and convex.

Heterogeneous To get the high performance, it is necessary to effectively use specialized hardware. But also within the single devices themselves we see specialization, and thus heterogeneity: the devices are made of multiple specialized logic chips, and a deep memory hierarchy. Even CPUs usually contain multiple levels of caching, a non-homogeneous communication structure or even multiple types of cores.

A simple example of the importance of using multiple specialized devices comes directly from an application of Amdahl's law [3]. Amdahl's law analyses the runtime of a parallel algorithm by dividing its operations into a serial part (that makes up a fraction s of the total number of operations) and a parallel fraction (that makes up $1 - s$ of the total). The serial part is made of interdependent

operations that must be done in a specific order and thus by a single processor. The parallel part is supposed to be perfectly parallelizable. In this way the speedup given by using P processors is:

$$S(P) = \frac{1}{s + \frac{1}{P}(1-s)} = \frac{P}{(P-1)s + 1}.$$

Therefore consider a Graphics Processing Unit (GPU), which is a highly parallel, throughput-oriented device, containing hundreds to many thousands of simple cores. These can accelerate the parallel part of an algorithm down to a very small time, as their scope is to process as much data as possible per unit time. But the single cores operate at a relatively low frequency. So the serial part comes to dominate the total runtime. The serial part should thus run as fast as possible, thus using a CPU, a latency-oriented device, designed to run a series of interdependent instructions in the least possible end-to-end runtime.

Sparse For some workloads, hardware specialization is indeed a viable path to more performance. This is especially important in some applications where there is a general structure that can be mapped into hardware. This is the case of dense linear algebra: hardware such as GPUs and Neural Processing Units (NPUs) are designed for this task. This type of workload is almost unique in the field of HPC, as memory is not always the limiting factor.

In many other cases the structure is less regular, and the development of focused hardware is ineffective, impractical or not economically viable. One such case is sparse linear algebra. In sparse linear algebra, the matrices and vectors have entries that are mostly zeros. This property is common when working with graphs, networks or in Finite Element Analysis (FEM). Therefore it is important to exploit the nonzero structure of the matrices in order to get fast algorithms for these specific tasks. But the nonzero structure can be highly variable, making it difficult to develop specialized hardware for such tasks. This is exacerbated by the inherently low operational intensity (i.e. amount of computation done per byte moved) of the task. This means that the memory costs (latency and bandwidth) quickly becomes the limiting factor. Being limited by memory movements is a problem common to many applications. This is such an important obstacle that it has been called ”memory wall” [13].

The two case studies in this thesis are based on distinct sparse linear algebra primitives, which are used to build efficient routines for more high-level tasks.

Thesis structure As anticipated, the thesis is structured around two case studies: in Chapter 2, we study highly parallel CPU-based implementations of two common optimization algorithms; in Chapter 3, we consider the implementation of a novel IR method for solving linear systems of equations in high precision, for Scientific Computing, running on Ascend NPUs.

Chapter 2

Parallel combinatorial optimization

Combinatorial optimization is a recurring High Performance Computing (HPC) workload, both as a target problem and as a component of larger scheduling, mapping, and design pipelines. Among its many formulations, Quadratic Unconstrained Binary Optimization (QUBO) and Ising models are especially relevant because they capture a broad class of binary decision problems while mapping naturally to linear algebra. They are also attractive in practice because heuristic solvers expose a direct tradeoff between time-to-solution and solution quality: by changing the number of iterations, replicas, or schedule parameters, one can tune accuracy to match a fixed compute budget. This is particularly useful on large sparse instances, where exact methods quickly become impractical.

The contribution presented in this chapter consists in the expression of known general-purpose QUBO and Ising heuristics using GraphBLAS primitives. Such implementation is appealing for HPC because their dominant kernels are Sparse Matrix–Vector products (SpMVs) and elementwise vector operations, which are among the central operations GraphBLAS backends optimize for. Furthermore, algorithms expressed using GraphBLAS can be easily ported to other platforms by using an implementation built for the selected hardware.

We focus on two complementary heuristics: Simulated Annealing-Replica Exchange (SARE), also known as Parallel Tempering, and Simulated Bifurcation (SB). SARE is naturally suited to Single Program Multiple Data (SPMD) execution because replicas can be updated independently between exchanges, while SB is dominated by repeated vector operations and matrix-vector products. Thus both algorithms fit well into the GraphBLAS programming model.

In principle these heuristics apply to any objective function $f : \{0,1\}^n \to \mathbb{R}$, but we focus on QUBO and Ising because of their practical importance and simple algebraic structure.

2.1 Background

2.1.1 QUBO and Ising problems

The Quadratic Unconstrained Binary Optimization (QUBO) and Ising problems are defined as follows.

Definition 2.1 (QUBO Problem)**.** Given a symmetric matrix $Q \in \mathbb{R}^{n\times n}$, the objective function is

$$f_{\mathrm{Q}}(x) = x^T Q x, \qquad x \in \{0,1\}^n,$$

and the optimization problem is to find a binary vector $x^\star \in \{0,1\}^n$ that minimizes $f_Q(x)$.

Definition 2.2 (Ising Problem)**.** In the Ising case, given a symmetric matrix $J \in \mathbb{R}^{n\times n}$ with zero diagonal and a vector $h \in \mathbb{R}^n$, the objective function is

$$f_{\mathrm{I}}(s) = s^T J s + h^T s, \qquad s \in \{-1,1\}^n,$$

and the optimization problem is to find a spin vector $s^\star \in \{-1,1\}^n$ that minimizes $f_{\mathrm{Ising}}(s)$.

The Ising Problem arises in Quantum Mechanics when considering a system as a set of interdependent particles characterized by a binary magnetic spin state.

Proposition 2.3 (Equivalence of QUBO and Ising)**.** *Given a symmetric matrix $Q \in \mathbb{R}^{n\times n}$, there exists a symmetric matrix $J \in \mathbb{R}^{n\times n}$ with zero diagonal, $h \in \mathbb{R}^n, C \in \mathbb{R}$ such that $\forall x \in \{0,1\}^n$:*

$$f_Q(x) = x^T Q x = s^T J s + h^T s + C = f_I(s) + C,$$

with $s_i = 2x_i - 1 \in \{-1,1\}\ \forall i \in \{1,2,\ldots,n\}$.

Proof. Let $e \in \mathbb{R}^n$ the vector with $e_i = 1 \ \forall i \in \{1, 2, \dots, n\}$; let $P \in \mathbb{R}^{n \times n}$ and $d \in \mathbb{R}^n$ be $P_{ij} = Q_{ij}$ for $i \neq j$, $P_{ii} = 0$ and $d_i = Q_{ii} \ \forall i \in \{1, 2, \dots, n\}$ the diagonal of Q. Then:

$$\begin{aligned} x^T Q x &= \frac{1}{4}(s+e)^T Q (s+e) = \frac{1}{4}\left(s^T Q s + e^T Q e + 2 s^T Q e\right) \\ &= \frac{1}{4}\left(s^T P s + d^T e + e^T Q e + 2 e^T Q s\right) \end{aligned}$$

Now we get the result by setting $J = \frac{1}{4}P$, $h = \frac{1}{2}e^T Q$ and $C = \frac{1}{4}\left(d^T e + e^T Q e\right)$. □

Many discrete optimization problems can be reduced to these forms through binary encodings, auxiliary variables, and penalty terms, making QUBO and Ising useful common targets for problem-agnostic heuristics. Among those, we highlight the Maximum cut (max-cut) problem [27].

Definition 2.4 (Max-cut problem)**.** Given an undirected graph $G = (V, E)$ with weighted adjacency matrix A, i.e. we denote with A_{uv} the weight on the edge $(u, v) \in E \subset V \times V$. Then the cut-value objective in binary variables is:

$$f_{\text{cut}}(x) = \sum_{(u,v) \in E} A_{uv}(x_u + x_v - 2x_u x_v), \qquad x \in \{0,1\}^{|V|},$$

where the term $x_u + x_v - 2x_u x_v$ equals 1 exactly when vertices u and v are assigned to different sides of the partition, and 0 otherwise. The max-cut problem is therefore to find a binary vector $x^\star \in \{0,1\}^{|V|}$ that maximizes $f_{\text{cut}}(x)$.

Proposition 2.5 (Equivalence of max-cut and Ising)**.** *For any graph G with weighted adjacency matrix $A \in \mathbb{R}^{n \times n}$, there exists a symmetric matrix $Q \in \mathbb{R}^{n \times n}$ and a constant C such that $\forall x \in \{0,1\}$:*

$$f_{\text{cut}}(x) = f_Q(x) + C.$$

Proof. We can suppose without loss of generality that A has zero diagonal, meaning that G has no self-loops. This does not change the size of any cut, since a self-loop may never connect vertices in different partitions. Observe that:

$$f_{\text{cut}}(x) = \sum_{(u,v) \in E} A_{uv}(x_u + x_v - 2x_u x_v) = 2x^T A x + (e^T A + e^T A^T)x$$

Now note that $(e^T A + e^T A^T)x = x^T \text{diag}(e^T A + e^T A^T)x$ since $x \in \{0,1\}^n$. Thus $Q = 2A + \text{diag}(e^T A + e^T A^T)$ makes $f_{\text{cut}}(x) = f_Q(x) \ \forall x \in \{0,1\}^n$. □

2.1.2 Related work

Recent work on QUBO and Ising optimization includes quantum-accelerated methods [23], physics-inspired solvers [38], and a broad range of classical heuristics [10]. This interest is driven in part by quantum and quantum-inspired computing platforms, which have made Ising and QUBO formulations common comparison targets. However, comparisons across such platforms often depend on embedding constraints, problem reformulations, and baseline choice, which can obscure the capabilities of strong sparse classical baselines. At the scales targeted by modern Ising and QUBO workloads, strong parallel scalability is essential, since without near-linear reductions in wall-clock time even strong classical baselines become impractical once exact methods are no longer viable. Many specialized approaches are most effective only after the problem has been adapted to a particular solver or accelerator through embedding, decomposition, or instance-specific tuning.

There has also been prior work on parallel simulated annealing on clusters [29, 9] and Graphics Processing Units (GPUs) [12], as well as on large-scale implementations of Parallel Tempering [24].

2.2 GraphBLAS

GraphBLAS is a specification for sparse linear algebra that defines a set of operations on sparse matrices and vectors [5]. The expected use-case of GraphBLAS is graphs: by expressing the graphs as (sparse) matrices, many common graph algorithms can be expressed as linear algebra operations [31].

We based the implementations on the ALP/GraphBLAS library [48, 30]. This library offers different backends. The ones we employ are: `reference` (single threaded), `reference_omp` (multithreaded) and `nonblocking` (nonblocking and multithreaded). The backend is selected at compile-time and implements the same algorithms and data structures while exploiting different optimization techniques. Nevertheless, the presented algorithms can be implemented using other libraries following the GraphBLAS specification [5].

In GraphBLAS, the primitives take operators as arguments. The library then takes into account the algebraic properties (e.g. associativity, distributivity) of the operator to generate optimized code. Some operators defined in the library are the following: `grb::right_assign` (right identity), `grb::add` (addition), `grb::mul`

(multiplication), `grb::leq` (less than or equal), `grb::neq` (not equal).

In the algorithms, we use the following ALP/GraphBLAS primitives:

- `grb::set(`a`,`b`)`, with two vectors of equal length as input, it assigns $a_i \leftarrow b_i$ for each index i. Alternatively b can also be a scalar, in which case it is implicitly broadcasted to a vector.
- `grb::foldl(`a`,`b`,`op`)`, which takes as inputs two vectors of equal length, and a binary operator op. It assigns $a_i \leftarrow op(a_i, b_i)$ for each index i where both a_i and b_i contain a value (i.e. are nonzero). Also in this case, if b is a scalar it is broadcasted to a vector.
- `grb::eWiseMul(`a`,`k`,`b`,`$ring$`)`, taking as input two vectors a, b of equal length, and a scalar k. The function assigns $a_i \leftarrow a_i + k \cdot b_i$. The last parameter, *ring*, corresponds to an algebraic semiring that defines which operations are used for $+$ and $\cdot$ in the preceding assignment. This parameter appears with the same function in the next primitives.
- `grb::dot(`s`,`a`,`b`,`$ring$`)`, takes as input a scalar s and two vectors a, b of equal length and modifies $s \leftarrow s + \sum_i a_i b_i$.
- `grb::mxv(`a`,`A`,`x`,`$ring$`)`, takes as inputs two vectors a, x and a matrix A of suitable dimension and assigns $a \leftarrow a + Ax$.

In our algorithms we used the *PlusTimes* semiring, corresponding to standard addition and multiplication.

These functions can also be used in a masked manner: in this case only entries of the output vector corresponding to entries of a mask vector m that are nonzero and evaluate to true are updated. The ALP/GraphBLAS library also supports mask inversion, denoted as $\neg m$, in which case the operation is applied on the explicit zeros of the mask. We will denote the masked variant of the primitives by overload of the unmasked version, e.g. `grb::foldl(`a`,`m`,`b`,`op`)`, `grb::set(`a`,`m`,`b`)`.

The primitives can take also a compile-time `descriptor` template parameter, which defines additional properties about the operation or its inputs. Mask inversion is done using a `descriptor`, thus using an inverted mask has no additional cost. Another `descriptor` we used is the `dense` descriptor. This descriptor is used when all input and output vectors are dense. This assumption enables automatic optimizations by simplifying and specializing the generated code to a dense structure.

2.3 Simulated Annealing-Replica Exchange

2.3.1 General Algorithm

The SARE algorithm for a general function $f : \{0,1\}^n \to \mathbb{R}$ is summarized in Algorithm 1 [28, 19].

The algorithm takes a number n_r of replicas (states) $r^{(1)}, \ldots, r^{(n_r)} \in \{0,1\}^n$ and the same number of inverse-temperatures $\beta^{(1)}, \ldots, \beta^{(n_r)} \in \mathbb{R}_{\geq 0}$. We will suppose that the inverse-temperatures are in decreasing order. The replicas are updated independently in a Simulated Annealing (SA) step and then exchanged.

The two steps are described in detail in Algorithm 1. Here we consider these procedures to be repeated for a fixed number of iterations. In general, the procedure can be halted earlier according to some convergence condition.

The exchange step (PT) stochastically exchanges neighboring replicas. The random value a is always ≥ 0, so if the lower energy replica is at higher temperature, the swap will always be accepted. Note that the order of exchange is important: by starting from the last replica and going down to the first, we ensure that low-energy replicas can go to low temperature quickly, and thus be optimized locally.

The SA step on a replica $r^{(i)}$ is similar in structure to the PT step, but works on the single replica: it considers changes to the individual variables, and accepts or rejects each of them individually with a probabilistic criterion. In this case an energy-decreasing change is always accepted. The order of update of the variables impacts the result as well, since the change to one variable will change the state for the subsequent variables. In this case, unlike PT, any order is equally effective.

In both cases, our algorithms are expressed using an exponential random variable, as that makes the GraphBLAS implementation simpler. This is non-standard, but it is equivalent to the commonly used acceptance criterion $u < e^{-\beta\Delta}$ with $u \in \text{Unif}(0,1)$:

$$\mathbb{P}(\Delta < a) = \min\{1, e^{-\beta\Delta}\} = \mathbb{P}(u < e^{-\beta\Delta}).$$

The same equivalence holds for PT, using $\beta = 1$ and δ instead of Δ.

Algorithm 1 The standard SARE algorithm.

Input: $n, n_r, N_{\text{iterations}} \in \mathbb{N}, \beta \in \mathbb{R}^{n_r}_{\geq 0}, \{r^{(i)} \in \{0,1\}^n \mid i \in \{1,\ldots,n_r\}\}, f : \{0,1\}^n \to \mathbb{R}$
Let $U \in \mathbb{R}^{n_r}$ be a vector such that $U_i = f(r^{(i)})$;
for $N_{\text{iterations}}$ times **do**
    for each replica $r^{(i)}$ **do**
        `SA(`$r^{(i)}$`,` $\beta^{(i)}$`,` f`);`
    end for
    `PT(`r`,` β`,` f`);`
end for
return lowest energy found, and relative replica
function PT(r, β, f)
    for $i = n_r$ down to 2 **do**
        Pick a random value $a \sim \text{Exp}(1)$
        $\delta \leftarrow \left(f\left(r^{(i)}\right) - f\left(r^{(i-1)}\right)\right) \cdot (\beta^{(i)} - \beta^{(i-1)})$
        if $-a < \delta$ **then**
            `swap(`$r^{(i)}, r^{(i-1)}$`)`
        end if
    end for
end function
function SA(r, β, f)
    $\delta_e \leftarrow 0$
    for each variable r_j in r **do**
        Pick a random value $a \sim \text{Exp}(\beta)$
        $\Delta \leftarrow f(r) - f\big((r_1,\ldots,1-r_j,\ldots,r_n)\big)$
        if $\Delta \leq -a$ **then**
            $r_j \leftarrow 1 - r_j$
        end if
    end for
end function

2.3.2 GraphBLAS implementation

The implementation of the SA step is described in Algorithm 2 using GraphBLAS primitives. This algorithm solves the QUBO problem, supposing the following input format: the symmetric matrix Q is expressed as $Q = \frac{1}{2}J + \text{diag}(h)$, where J is a symmetric matrix with zero diagonal and h is a vector, thus minimizing $f_Q(x) = x^T Qx = \frac{1}{2}x^T Jx + x^T h$ with $x \in \{0,1\}^n$.

Algorithm 2 The SA step expressed using GraphBLAS primitives. Objects prefixed with `grb::` are standard library objects, others are custom-defined.

function SAGRB(h_0, x, J, β)
  Let `masks` be a set of independent masks forming a partition of J.
  Let $s \in \mathbb{R}^n$ be a vector of iid random variables $-s_i \sim \text{Exp}(\beta)$
  $\delta_e \leftarrow 0$
  `grb::set(`h`,` h_0`)`
  `grb::mxv(`h`,` J`,` x`,` *PlusTimes*`)`
  for *mask* in `masks` **do**
    `grb::set(`dn`,` *mask*`,` x`)`
    `grb::foldl(`dn`,` $\neg x$`,` -1`, grb::right_assign)`
    `grb::foldl(`dn`,` h`, grb::mul)`
    `grb::foldl(`dn`,` *mask*`,` s`, grb::leq)`
    `grb::set(`*accept*`,` dn`,` *mask*`)`
    `grb::foldl(`x`,` *accept*`,` 1`, grb::neq)`
    `grb::set(`δ`,` *accept*`,` x`)`
    `grb::foldl(`δ`,` $\neg$*accept*`,` -1`, grb::right_assign)`
    `grb::dot(`δ_e`,` δ`,` h`,` *PlusTimes*`)`
    `grb::mxv(`h`,` J`,` δ`,` *PlusTimes*`)`
  end for
  return δ_e ▷ This value is needed to keep track of $f_Q(r)$ and avoid its recomputation.
end function

Masking The implementation of the SA step uses a partitioning of the variables into masks. The set of masks `masks` $\subset \{0,1\}^n$ is constructed such that they are independent i.e. $\forall m \in$ `masks`$, m^T Qm = 0$ and form a partition of the variables i.e. $\sum_{m \in \texttt{masks}} m$ is the vector of all ones. The change to one variable in a mask does

not affect the change of energy by flipping another variable in the same mask. Therefore the updates of variables in a mask can be done concurrently.

The construction of such masks is equivalent to the problem of graph coloring on a graph with Q as adjacency matrix (ignoring self-loops). For this task we used a randomized greedy algorithm expressed in GraphBLAS [35, Algorithm 2]. The idea of this algorithm is the following: first assign a random value to each node; then iteratively consider nodes whose neighbors either have already been assigned a color, or have larger value. These nodes are assigned the lowest possible color.

Multiprocess implementation

It is natural to express the SARE algorithm using an SPMD paradigm. In the SA step each of the replicas is updated independently, making this step embarrassingly parallel. Thus we can partition the replicas among multiple independent processes, while keeping the implementation of the SA step unchanged, as stated in Algorithm 2.

Neighboring replicas may be in different memory spaces, so the PT step has to be changed as described in Algorithm 3. The pseudo-code employs buffered one-sided RDMA (Put), and assumes synchronisation has completed before the first use of the communicated variable at the remote location. In the implementation, we use `MPI_Send/Recv`. Therefore the processes are loosely coupled, with a 1d wavefront communication scheme, as shown in Figure 2.1.

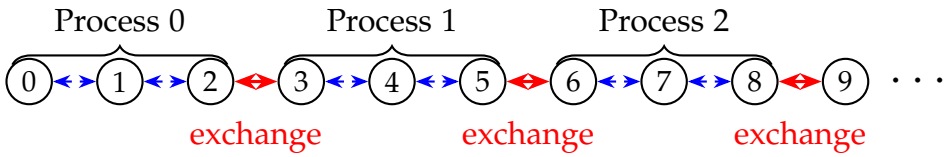

Figure 2.1: Diagram showing the replicas (circles), and their division between processes, and required communication between them. In blue are local exchanges, in red exchanges between different processes.

While in the shared-memory case an exchange can be done in constant time, exchanging replicas between different processes requires $\mathcal{O}(n)$ time. This is amortized by the fact that the replicas are not always exchanged, and so it may be sufficient to communicate the energies and the betas to evaluate δ (requiring

Algorithm 3 Pseudocode for the PT step in the multiprocess implementation. Here $m = \lceil n_r / P \rceil$. Replicas $pm + 1, \ldots, p(m+1)$ are local to process p.

function PTSPMD(r, β)
  Let $p \in \{0, \ldots, P-1\}$ be the rank of the current process.
  if $p < P - 1$ **then**
    Pick a random value $a_p \sim \mathrm{Exp}(1)$
    $j \leftarrow p(m+1)$
    Put $f_{\mathrm{Q}}\left(r^{(j)}\right), \beta^{(j)}, a_p$ into process $p+1$
    $\delta \leftarrow \left(f_{\mathrm{Q}}\left(r^{(j+1)}\right) - f_{\mathrm{Q}}\left(r^{(j)}\right)\right) \cdot (\beta^{(j+1)} - \beta^{(j)})$
    if $-a_p < \delta$ **then**
      Put $r^{(j)}$ into process $p+1$
      $r^{(j)} \leftarrow r^{(j+1)}$
    end if
  end if
  PT(r, β, f_{Q})
  if $p > 0$ **then**
    $j \leftarrow pm$
    Put $f_{\mathrm{Q}}\left(r^{(j+1))}\right), \beta^{(j+1)}$ into process $p-1$
    $\delta \leftarrow \left(f_{\mathrm{Q}}\left(r^{(j+1)}\right) - f_{\mathrm{Q}}\left(r^{(j)}\right)\right) \cdot (\beta^{(j+1)} - \beta^{(j)})$
    if $-a_{p-1} < \delta$ **then**
      Put $r^{(j+1)}$ into process $p-1$
      $r^{(j+1)} \leftarrow r^{(j)}$
    end if
  end if
end function

only constant bandwidth). Nevertheless this is asymptotically negligible, since ideally the exchange probability is constant.

Hybrid implementation

Thanks to the use of the GraphBLAS library, the multiprocess implementation can be made hybrid simply by compiling with a multithreaded backend: the MPI and multithreading infrastructure work together. This is a simple way to get additional speedup in the case every process already handles a single replica, and thus no more multiprocessing can be applied [1].

2.3.3 Theoretical Analysis

Let P be the number of processes; let Q be an $n \times n$ matrix with n_{nz} nonzeros. We analyze first the runtime of a single process, then the time required by the communication between them.

Computation and local movements The total work and memory movements are of the same order: $\mathcal{O}(n + n_{nz})$ in Algorithm 2, and $\mathcal{O}(n + n_r)$ in Algorithm 3. The n_r replicas are balanced between the processes, so the runtime of each process is $T_{comp}(P) = \Theta\left(\frac{n_r}{P}(n + n_{nz}) + n\right)$ at each iteration. The total runtime is just the time for a single iteration multiplied by the number of iterations. Therefore for simplicity we will consider a single iteration.

Using this bound, we can estimate the speedup:

$$S(P) = \frac{T_{comp}(1)}{T_{comp}(P)} = \Theta\left(\frac{n_r(n + n_{nz})}{\frac{n_r}{P}(n + n_{nz}) + n}\right) = \Theta\left(P\frac{n_r(n + n_{nz})}{n_r(n + n_{nz}) + nP}\right).$$

In other words the efficiency is:

$$E(P) = \frac{S(P)}{P} = \Theta\left(\frac{n_r(n + n_{nz})}{n_r(n + n_{nz}) + nP}\right).$$

[1] There is a "good" number of replicas: a problem with N variables is optimized well with approximately $\sqrt{N}$ replicas [11]. Having more replicas than this does not improve the solution quality.

If we suppose that the workload is balanced, with each process holding $K \in \mathbb{N}$ replicas, that is $n_r = KP$, we get:

$$\tilde{E}(K) := E\left(\frac{n_r}{K}\right) = \Theta\left(\frac{K(n+n_z)}{n+K(n+n_z)}\right). \tag{2.1}$$

Which is independent of P. This is reasonable, since the number of operations each process does is determined only by K and not P nor n_r.

We can also predict an asymptotic upper bound on the strong scaling, as function of the number of replicas and density of the instance:

$$\lim_{P\to\infty} S(P) = \lim_{P\to\infty} \Theta\left(\frac{n_r(n+n_{nz})}{\cancel{n_r(n+n_{nz})/P} + n}\right) = \Theta\left(n_r\left(1+\frac{n_{nz}}{n}\right)\right).$$

This result suggests that the density of the matrix Q is the more important predictor of efficiency, with denser matrices yielding better efficiency.

The memory requirement is $\mathcal{O}(n_r n + n_{nz})$ per process, independently of the number of iterations. In fact the vectors are reused, and a single matrix is allocated and reused for the different replicas.

Communication Only Algorithm 3 involves communication. Here we'll denote $r^{(i)}_{(j)}$ the i-th replica at iteration j. The communication pattern is described in Figure 2.2. $r^{(i)}_{(j)}$ for $i > 1, j > 0$ has the following dependencies:

- $r^{(i)}_{(j-1)}$ through the SA step, which is local to each process;
- $r^{(i-1)}_{(j-1)}$, as $r^{(i)}_{(j)}$ might be exchanged with it before starting the next iteration;
- $r^{(i+1)}_{(j)}$, as $r^{(i)}_{(j)}$ might get exchanged with it at the start of the PT step.

This dependency graph is isomorphic to that of a 1-dimensional stencil of radius 1.

To quantify the total time needed for communication, we can look at two costs: bandwidth cost W (how many words of data are moved) and synchronization cost (maximum number of messages sent by any single process). Combining these values with the bandwidth β and latency γ of a system, we can estimate the communication time T_C as:

$$\max\{W\cdot\beta, S\cdot\gamma\} \leq T_C \leq W\cdot\beta + S\cdot\gamma.$$

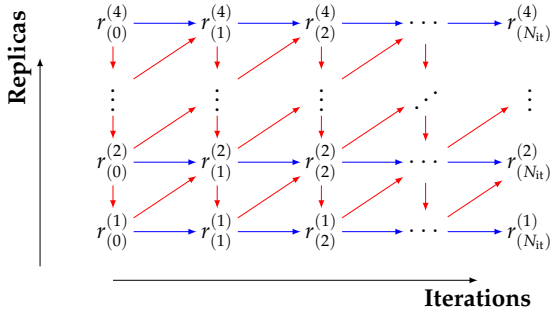

Figure 2.2: Diagram describing the data dependencies between the replicas at different iterations. Red arrows indicate swaps that may need communication between processes. Blue dependencies do not need communication.

From the diagram in Figure 2.2, it is easy to see that the critical path for the algorithm is of length $n_r + N_{\text{iterations}}$. Therefore this is a lower bound for the synchronization cost of the whole algorithm. Furthermore, the synchronization cost incured by a single process is constant at each iteration: only up to 4 synchronizations are required in Algorithm 3, 8 for blocking message-passing. Therefore the total synchronization cost for the presented implementation is at most $\max\{n_r + N_{\text{iterations}}, 8N_{\text{iterations}}\}$.

The bandwidth cost can be estimated as $\mathcal{O}(N_{\text{iterations}}n)$, as only up to $\mathcal{O}(n)$ data is communicated between each pair of neighboring processes at each iteration to exchange a replica between distinct memory spaces.

2.4 Simulated Bifurcation

2.4.1 General Algorithm

Simulated Bifurcation is a physics-inspired technique to optimize an Ising system, based on the adiabatic evolution of a classical nonlinear system. For the details of the algorithm, we refer to Goto et al. [14]. The idea is to simulate an Ising machine using a symplectic Euler method following the Hamiltonian equations of motion. The algorithm starts with a position $x^{(0)} \in [-1, 1]^n$ and momentum $y^{(0)} \in [-1, 1]^n$. Then the position and momentum are updated iteratively. The position is updated as:

$$x^{(k+1)} = x^{(k)} + \Delta_t a_{\max} y^{(k+1)},$$

where Δ_t and $a_{\max} > 0$ are scalar parameters. After this update, if for some i, $|x_i^{(k+1)}| > 1$, we set $x_i^{(k+1)} = \text{sign}(x_i^{(k+1)})$ and $y_i^{(k+1)} = 0$ (perfectly inelastic walls).

For the update of the momentum, Goto et al. [14] describe multiple options. The best performing is discrete Simulated Bifurcation (dSB), in which the discretized state $s_i^{(k)} = \text{sign}(x_i^{(k)}) \in \{-1, 1\}$ is used in the update of the momentum:

$$y^{(k+1)} = y^{(k)} + \Delta_t \left((a_k - a_{\max}) x^{(k)} + c_0 \left(J s^{(k)} + h \right) \right).$$

The values Δ_t, the number of iterations $N_{\text{iterations}}$, $a_{\min}$, $a_{\max}$ and c_0 are scalar parameters; these determine $a_k = a_{\min} + \frac{k}{N_{\text{iterations}}}(a_{\max} - a_{\min})$. We will discuss the specific settings of these parameters in Section 2.5.1.

2.4.2 GraphBLAS implementation

The implementation of dSB, expressed using GraphBLAS primitives, is described in Algorithm 4. The `dense` descriptor is applied to all operations in the algorithm, as all vectors are dense.

It is worth noting that in the translation of dSB into GraphBLAS, we reuse the matrix–vector product Js from the computation of the energy for the momentum update in the following iteration.

To get the best performance, we express some functions by defining some custom operators. In this way we can express with a single primitive, operations that would otherwise require multiple steps, automatically fusing kernels and improving memory locality. The custom operators are the following:

$$\texttt{Sign}(x, y) = \begin{cases} -1 & \text{if } y < 0 \\ 1 & \text{if } y \geq 0 \end{cases}$$

$$\texttt{AbsGtAssign}(x, y) = \begin{cases} x & \text{if } |y| \leq 1 \\ 0 & \text{if } |y| > 1 \end{cases}$$

$$\texttt{Clip}(x, y) = \begin{cases} y & \text{if } |y| \leq 1 \\ y/|y| & \text{if } |y| > 1 \end{cases}$$

Algorithm 4 The dSB algorithm expressed using GraphBLAS primitives. Objects prefixed with `grb::` are standard library objects, others are custom-defined.

Input: $x, y, h \in \mathbb{R}^n, J \in \mathbb{R}^{n \times n}$, scalar parameters $\Delta_t, c_0, a_{\min}, a_{\max}$
Let $tmp \in \mathbb{R}^n$ and $s \in \{-1, 1\}^n$ be temporary vectors of length n
`grb::set(`tmp`, 0)`
`grb::set(`s`, 1)`
`grb::foldl(`s`, `x`, Sign)`
`grb::mxv(`tmp`, `J`, `s`, PlusTimes)`
for k from 0 to $N_{\text{iterations}}$ **do**
    $a_k \leftarrow a_{\min} + (a_{\max} - a_{\min}) \cdot k / N_{\text{iterations}}$
    `grb::foldl(`tmp`, `h`, grb::add)`
    `grb::eWiseMul(`y`, `$\Delta_t c_0$`, `tmp`, PlusTimes)`
    `grb::eWiseMul(`y`, `$\Delta_t(a_k - a_{\max})$`, `x`, PlusTimes)`
    `grb::eWiseMul(`x`, `$\Delta_t a_{\max}$`, `y`, PlusTimes)`
    `grb::foldl(`y`, `x`, AbsGtAssign)`
    `grb::foldl(`x`, `x`, Clip)`
    `grb::foldl(`s`, `x`, Sign)`
    `grb::set(`tmp`, 0)`
    `grb::mxv(`tmp`, `J`, `s`, PlusTimes)`
    $d_{Js} \leftarrow 0, d_h \leftarrow 0$
    `grb::dot(`d_{Js}`, `tmp`, `s`, PlusTimes)`
    `grb::dot(`d_h`, `h`, `s`, PlusTimes)`
    $f_{\text{I}}\left(s^{(k)}\right) \leftarrow \frac{1}{2} d_{Js} + d_h$
end for
return $f_{\text{I}}\left(s^{(N_{\text{iterations}})}\right), s$

2.5 Experimental Evaluation

2.5.1 Testing methodology

We evaluated our implementation on the systems summarized in Table 2.1. To test scaling beyond a single node, we employed multiple ARM and Cascade nodes. We always pinned the processes and threads to distinct physical cores, therefore the use of multiple nodes will be clear by the number of processes and threads used compared to the single node. Note that in the case of Intel Central Processing Units (CPUs) hyperthreading was enabled, but only a single thread for each core was actually used. The interconnect between nodes is a 100 Gbps Infiniband network with measured latencies of around 30 µs.

The ARM and Cascade machines run Ubuntu 22.04, Linux kernel 5.15.0 and MPICH 4.0; the Comet machine runs Ubuntu 24.04, Linux kernel 6.8.0 and MPICH 4.2.0. The compiler used was GCC, version 11.4 on Cascade and ARM, and version 13.3 on Comet. Compilation was done using the following flags: `-O3 -march=native -mtune=native -funroll-loops`. OpenMP affinity policy was always set to `close`.

Name	CPU	Cores	Sockets
Comet	Intel Xeon W-1270	8	1
ARM	HiSilicon Kunpeng-920	96	2
Cascade	Intel Xeon Gold 6238T	44	2

Table 2.1: Systems used for benchmarking.

Baselines

We compared our implementations to the following state-of-the-art sequential solvers:

- `dwave-neal`, a state-of-the-art SA implementation from the `dwave-samplers` package [21].

  SA mimics thermal cooling by performing stochastic single-spin flips at a decreasing temperature to escape local minima. This is in contrast to our implemented algorithm SARE, which runs multiple replicas of the system at different temperatures, periodically swapping configurations to enhance

exploration. While both methods have rigorous theoretical foundations, they offer only weak practical convergence guarantees, as extremely slow cooling schedules are required to ensure global optimality [18].

In practice, faster geometric schedules are typically used. SARE tends to explore rugged energy landscapes more robustly than SA by allowing low-temperature replicas to leap to high-temperature states via swaps [4], though this requires tuning multiple temperatures. Recent studies [44] confirm that SARE can outperform SA in both solution quality and time-to-solution on hard spin-glass benchmarks, despite the increased computational cost. Conversely, SA is simpler and can be more efficient for easier problems.

- MQlib [10], a meta-heuristic framework that selects a solver among a selection of algorithms based on input problem characteristics. The set of solvers includes Tabu search, Genetic Algorithms, Local Search techniques, and Simulated Annealing.

Benchmark problems

Following established practices in this field, we evaluated our implementation using Max-Cut problems. The conversion into QUBO or Ising formulation is done following the proof of Theorem 2.5.

The benchmark graphs are drawn from the **Gset collection** [47]. For the performance data we considered the following larger set of instances: G1, G11, G21, G22, G31, G41, G51, G61, G81. In this way, we capture the performance on a variety of sizes, as shown in Table 2.2.

Nevertheless goal is not to average over all problem classes, but to compare implementation quality in a regime where the selected methods perform well. Therefore we highlighted G22 and G81 because they are highly frustrated sparse instances where SARE is known to perform well.

Solver settings

For the SARE experiments, we used a fixed geometric temperature gradient across the replicas, ranging from $\beta^{(1)} = 10^2$ to $\beta^{(n_r)} = 10^{-3}$. While this may limit the solution quality compared to adaptive schemes, it provides a consistent basis for evaluation. In principle, adaptive temperature techniques could

Table 2.2: Basic properties of the Gset graphs used for testing.

Name	n	**Number of edges**	**Type**
G1	800	19176	random
G11	800	1600	toroidal
G21	800	4667	planar
G22	2000	19990	random
G31	2000	19990	random
G41	2000	11785	planar
G51	1000	5909	planar
G61	7000	17148	random
G81	20000	40000	random

be integrated into our solver, since information about neighboring replicas is already exchanged, such schemes would not require additional communication and thus would probably not significantly impact performance. However, as this work focuses primarily on computational performance, we did not explore these adaptive methods further.

For the number of replicas n_r, we always considered cases where n_r is a multiple of the number of processes P: in this way the workload is balanced between the processes. In the tests we run SARE for 400 iterations for quality tests, and 200 iterations for the other tests. The number of replicas is selected to be n_r is twice the number of physical cores: 192 for ARM nodes, 88 for Cascade nodes, and 16 for Comet. For the multinode tests we consider also n_r to be four times the number of physical cores.

For dSB the scalar parameters $\Delta_t, a_{\min}, a_{\max}, c_0$ were fixed to the following values: $\Delta_t = 0.75, a_{\min} = 0, a_{\max} = 1, c_0 = \frac{1}{2\langle J\rangle\sqrt{n}}$, where $\langle J\rangle = \sqrt{\frac{\sum_{i,j} J_{i,j}^2}{n(n-1)}}$. Motivation for these choices is provided in [14].

In all cases the starting states were initialized as elementwise iid samples: $r_j^{(i)} \sim Ber(0.5)$ for SARE, and $x_i^{(0)}, y_i^{(0)} \sim Unif([-1,1])$ for SB. The number of iterations for dSB is always 10000.

In all cases, the experiments were repeated for 5 warmup runs and then 100 timed runs.

2.5.2 Results

In general measured runtimes were fairly concentrated, with a standard deviation under 1% for single node runs and up to 5% in the multinode case.

Unless specified, the tests use the `reference` backend, to avoid the effects of multithreading and only measure the effect of the intended parameters. Multithreaded backends are evaluated in specific sections.

Solution Quality

To validate solution quality and single-threaded efficiency, we compare our implementations against state-of-the-art solvers on the instances described in Section 2.5.1. All experiments in this section use a single process and thread on the Comet machine.

The time comparisons in Figure 2.3 are intended as a proxy for the computational work required to reach a given solution quality, not as absolute performance limits: all solvers could run faster on larger machines or with better tuning. Within this controlled setting, the following observations hold.

There is a meaningful difference in quality between `dwave-neal`, MQLib and GraphBLAS SARE. This is definitely to be expected, as the algorithms are the different behavior of simulated annealing and parallel tempering. As observed in Section 2.5.1, the temperature gradient of SARE is simple and fixed, so this is a possible explanation for the slightly worse solution quality.

The quality of dSB is in line with the results reported by Goto et. al [14], that we were able to reproduce. The runtime reported in that work and what we measured for our implementation was comparable, and slightly in favour of our solver on the large sparse instances. A precise comparison doesn't make sense because of the use of very different hardware and their batched approach.

Performance impact of using masks

The SARE implementation uses masking to improve memory locality. To assess whether this technique indeed yields better performance we compare the presented implementation with the same algorithm, but using the basis vectors $e_1, e_2, \ldots, e_n$ as masks instead of computing masks with multiple independent variables.

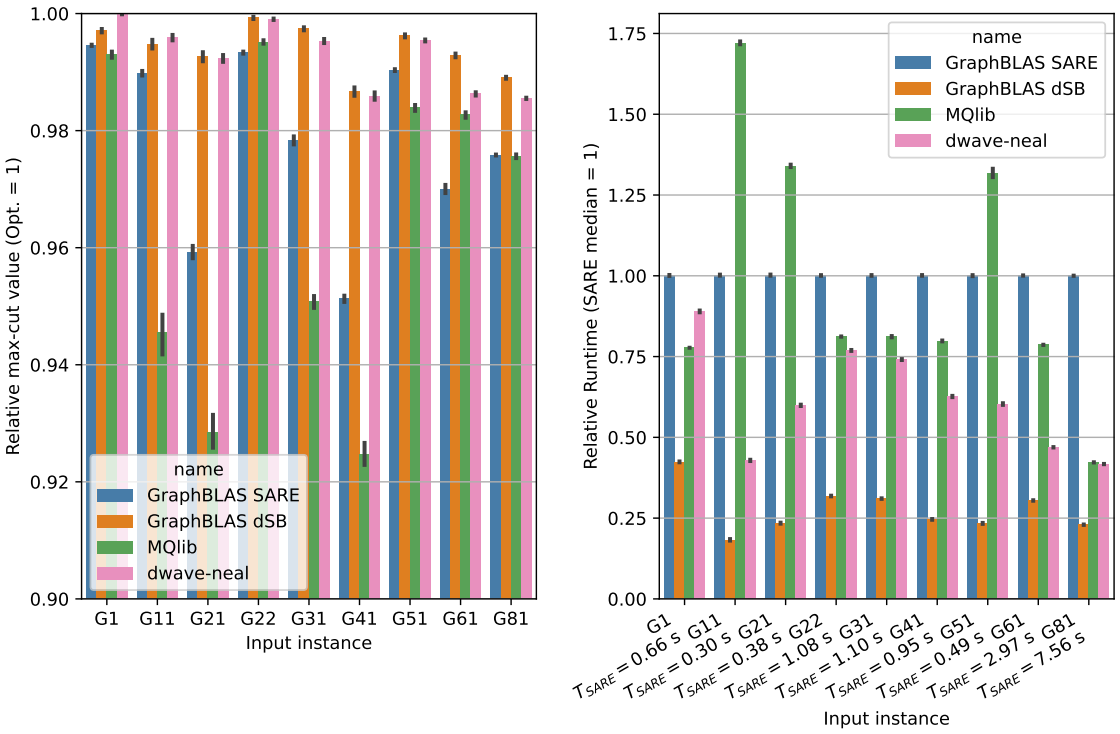

Figure 2.3: Single-threaded solution quality and (relative) runtime for different solvers and instances. Aggregate data of 100 runs with different random seeds. dwave-neal and SARE used 400 iterations; MQLib had a fixed runtime budget set close to the measured runtimes of the other solvers. Error bars are 95% confidence intervals.

The comparison of the runtimes for the two versions is shown in Figure 2.4. The results clearly show a speedup across all tested instances on both ARM and Cascade. The speedup is up to 5 times for the smaller instances, but significantly larger (up to 23) for G61 and G81.

There are probably two competing reasons at play here: one is cache size and the other is matrix density. A higher matrix density (such as for G1, G22 and G31) in general means sparser masks, that in turn makes the masking implementation more similar to the unmasked one. Conversely if a graph is smaller, such as G11, G21 or G51, more of the working data fits in cache. In this case increased memory locality doesn't make much of a difference since in both cases the working data never goes out of the cache and can be accessed fast.

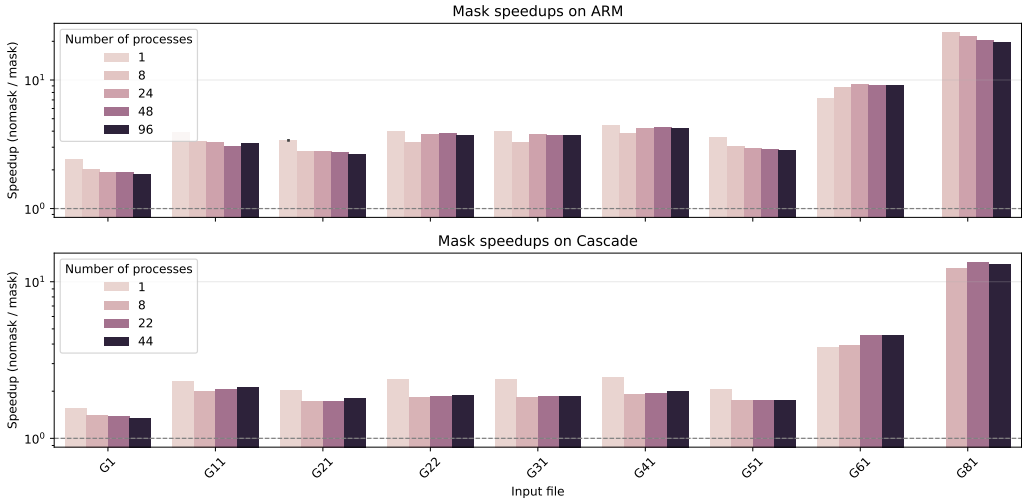

Figure 2.4: Speedup given by masking use on different Gset instances on ARM and Cascade nodes. Unmasked runs on G81 with a single process took over 30 minutes each, and thus it was unfeasible to have a reliable estimate.

Strong scaling of multiprocess SARE

In this section we evaluate the strong scaling of multiprocess SARE: we fix the workload parameters (n_r) and see how the performance change when increasing the computational resources, the number of processes in this case. The efficiency plots for the results of these tests are shown in Figure 2.5.

The results show that there is a consistent efficiency when operating on a single node and with a number of processes up to the number of physical cores; also with up to 4 nodes the runtime decreases further, but with markedly lower efficiency. The test suggest that the multiprocessing overhead is lightly affected by the number of processes, while clearly depending on the latency between them. In fact the efficiency is relatively constant within a single node, while significantly decreasing for multiple nodes.

Looking at the difference between the runs with 192 and 384 replicas (2x and 4x node size respectively) on ARM, we see a higher efficiency for the second. This is exactly what was suggested by Equation (2.1). The drop in performance on the rightmost points is probably due to the inefficiency of each process handling a single replica. Another view on these effects comes from the following analysis of weak scaling.

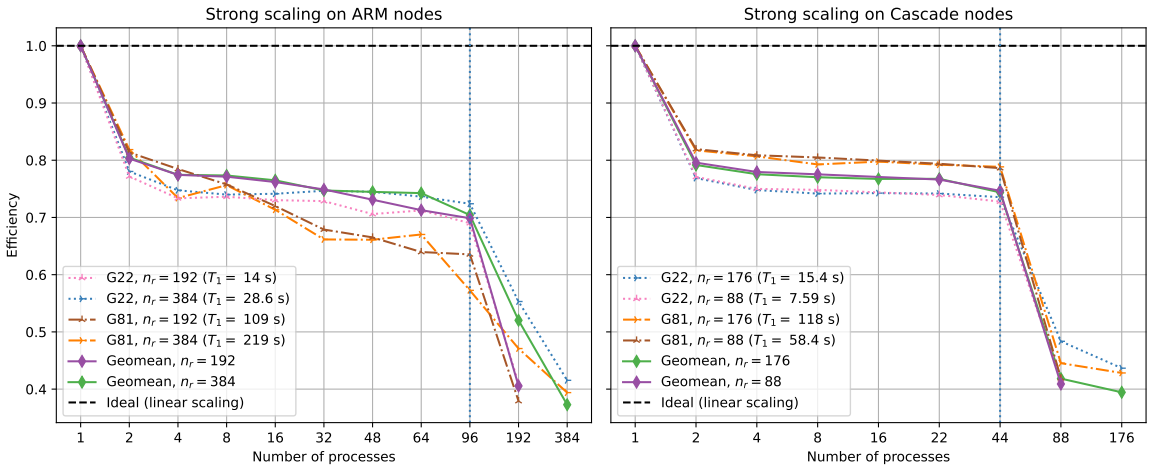

Figure 2.5: Multiprocess efficiency (speedup/number of processes) with a fixed workload using the `reference` backend. Single process time T_1 is in label. Vertical dotted line indicates single-node core count. For each graph, the median of 100 runs is considered.

Weak scaling of SARE

To test weak scaling, we fix number of iterations and number of replicas per process and vary the number of processes. Thus the total number of replicas changes proportionally to the number of processes. The resulting efficiency for these tests is shown in Figure 2.6.

The efficiency is basically constant as we increase the number of processes, but with a significant difference for the single- and multi-node cases. This confirms that the communication overhead for each process is independent of the number of total replicas, as predicted in Section 2.3.3.

The efficiency drop also suggests that the communication performance is an important factor for the efficiency. More specifically the latency seems to be more important than the bandwidth, since the difference in efficiency between G22 ($n = 2\,000$) and G81 ($n = 20\,000$) in multinode cases is relatively small, especially for Cascade nodes.

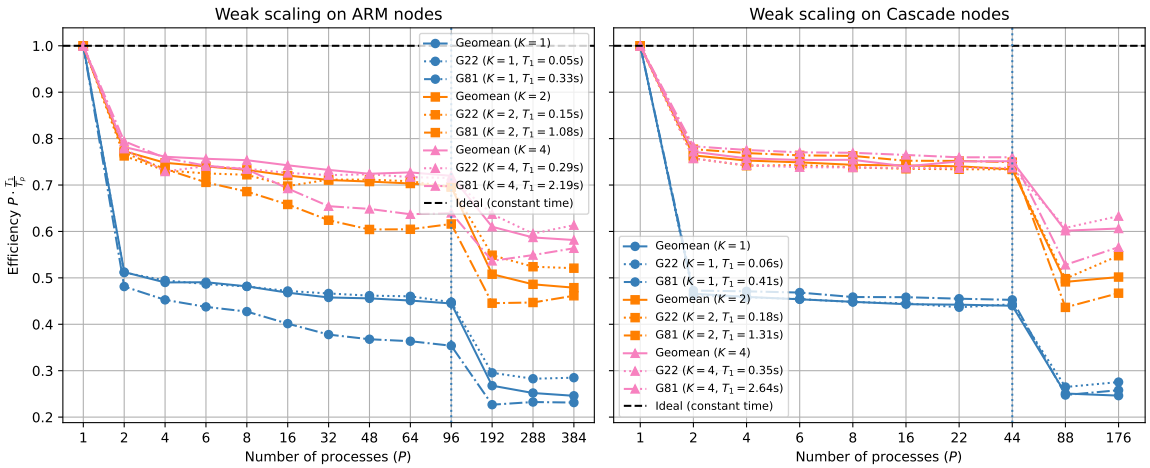

Figure 2.6: Plots showing weak scaling efficiency of the SARE algorithm with the reference backend on Gset graphs using ARM nodes with different numbers of replicas per process K.

Hybrid SARE

To effectively run concurrently the multithreaded backends, it was necessary to pin the processes and the threads to hardware cores. When this was not done, the performance of multithreaded versions would be significantly worse.

Given the large number of configurations to test, the tests for hybrid SARE we concentrate on G81. The median runtimes for this case are reported in Table 2.3, together with the results of the multiprocess implementation analyzed above, that uses the `reference` backend.

In the `nonblocking` backend, a small collective is run before every execution to assess the sparsity structure of the vectors. This operation introduces significant overhead for the matrices used. This explains why the `nonblocking` is consistently slower than `reference`, even with multiple threads.

On the other hand, the `reference_omp` backend shows some overhead compared to reference, but also a faster runtime using multiple threads. Nevertheless it is more efficient to increase the number of processes than threads: speedup 1.7–2.0 for twice the number of processes vs. 1.2–1.5 for doubling the number of threads.

Table 2.3: Median runtimes out of 100 runs for hybrid SARE on G81 on the ARM machine (values in seconds). Missing values are due to either exceeding the number of cores in a single node, or runtime exceeding 200 seconds.

Backend		`reference_omp`				`nonblocking`			`ref.`
		Threads per process				Threads per process			
		1	2	4	6	1	2	4	
Processes	1	131	102	79.1	86.8				109
	2	79.3	62.6	45.2	43.1	189	148	126	67.3
	4	41.4	32.0	23.2	23.4	78.9	57.0	45.4	35.1
	8	21.7	16.5	11.5	11.4	35.3	23.9	18.4	17.9
	16	12.1	8.63	5.85	5.81	17.2	10.9	7.87	9.43
	32	6.61	4.54			9.00	5.23		5.01
	48	4.63	3.07			6.15	3.44		3.39
	96	2.39				3.72			1.75

Discrete Simulated Bifurcation

The dSB algorithm consists of a simple loop of dense vector operations and sparse matrix-vector products. Therefore its performance depends almost entirely on the underlying GraphBLAS implementation in this favorable regime.

Figure 2.7 shows the speedup that dSB achieves with different number of threads. The data suggests that the size of the instance impacts efficiency: while G22 and the geometric mean show little speedup, for the larger G81 we have good speedup up to 6 threads with both mutlithreaded backends. From then on, the speedup plateaus or even decreases for both tested backends, as the memory bandwidth becomes the limiting factor.

In fact, a simple roofline estimate for the Cascade node using an L2 cache bandwidth value of 629 GB/s [42] (this is likely an overestimate) and an arithmetic intensity of 0.25 flops/byte (working in single precision), we get an upper bound of around 158 Gflop/s. But each core achieve a peak of around 30 Gflop/s, so we the algorithm becomes memory bound with 6 threads, which is consistent with the observed results for `reference_omp`.

The `nonblocking` backend outperforms `reference_omp`. The primary reason for this is that, by fusing operations, it increases the cache utilization, helping utilize better the limited memory bandwidth. Another reason for better performance, especially on the smaller instances and with many threads, is that

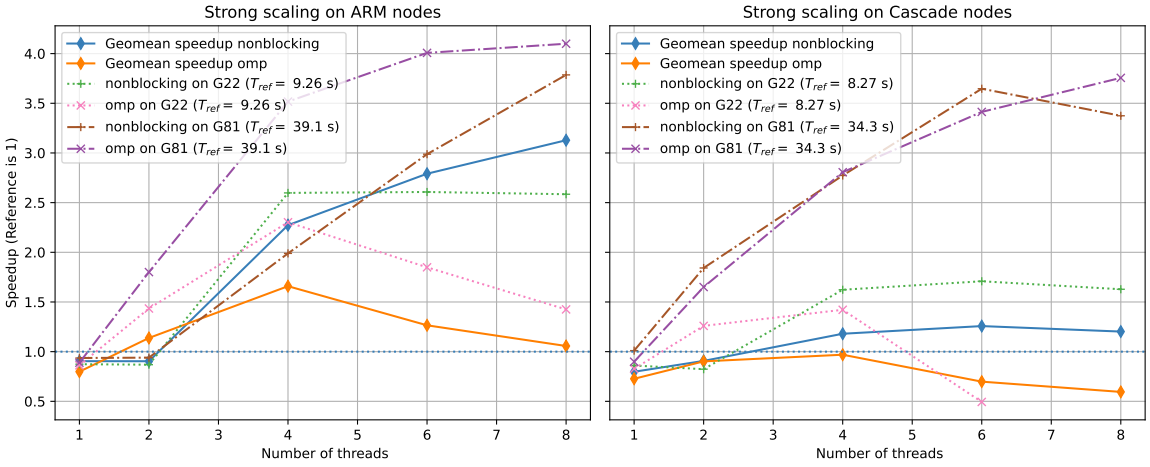

Figure 2.7: dSB speedup versus thread count. Time using `reference` backend T_{ref} is in label.

`nonblocking` automatically limits the number of threads and other runtime parameters according to a performance estimate [30]. It is also noteworthy that in one case (G81 on Cascade) the `nonblocking` backend shows slightly faster single-threaded performance than `reference`.

2.6 Conclusion

In this chapter we presented some algorithms for optimization, implemented using GraphBLAS. The two algorithms, SARE and dSB, exhibit different scalability performance, that is dictated by the characteristics of the algorithms.

SARE uses MPI to offer large-scale scalability, also across multiple nodes, thanks to the embarrassingly parallel nature of the SA step. The implementation shows good efficiency when each individual process handles more than one replica, thus having a good work-communication balance. The results also suggest that the efficiency is significantly affected by the interconnect latency as the multi-node efficiency is consistently lower than the single-node case.

On the other hand, dSB is limited to a single process, though in a real use-case the solver can be run in multiple concurrent searches. The algorithm maps naturally to GraphBLAS primitives, thus easily exploiting the optimizations in the underlying library. This is effective also thanks to the use of only dense

vectors, which are handled effectively by the `nonblocking` backend.

Chapter 3

Iterative Refinement on Neural Processing Units

In recent years there has been a clear trend in hardware design: the throughput for low-precision floating point formats increased significantly, while it stagnates for high-precision ones. This has been accelerated by the requirements of large-scale Artificial Intelligence (AI) workloads, which have become a driving factor in hardware design with many new designs being put forward in this direction [26, 2].

In this chapter we consider the problem of solving a sparse linear system. Given a sparse matrix $A \in \mathbb{R}^{n \times n}$, and a vector $b \in \mathbb{R}^n$ the task is to find $x \in \mathbb{R}^n$ such that $Ax = b$. The challenge is to combine in the algorithm a scheme that emulates high-precision computations using the limited data type support of Neural Processing Units (NPUs).

A possibility is to use an Ozaki Scheme (OS) (described in Section 3.1.3) to emulate high-precision operations, and run a standard algorithm. Nevertheless, such method doesn't exploit the additional structure neither of the OS in the solver or, vice-versa, of the solver in the OS. Therefore, the aim of this work is to vertically integrate the method: by exploiting and vertically integrating the structure, the hope is to get a fast solver, competitive with standard Central Processing Unit (CPU) algorithms.

We describe the novel method in Section 3.2; but first, we formalize, detail and explore some useful preliminaries in Section 3.1; finally we evaluate experimentally the proposed method in Section 3.3.

3.1 Background

3.1.1 Representation of floating point numbers

The traditional way to represent floating point numbers is described in the IEEE 754 specification [20], which is implemented in virtually all existing hardware.

The idea is to use scientific notation in binary: each number is represented by a single sign bit S, an exponent E and a mantissa M. Figure 3.1 exemplifies the representation for the single precision case (32 total bits).

The exponent E is an integer represented as a n_e-bit unsigned integer containing the binary value $E + B$. The bias B is a fixed value set to $2^{n_e} - 1$.

The mantissa can be seen as a fractional binary value in $[1, 2)$. But only the bits after the decimal point are stored, while the one that comes before the point is implicit[1].

The value of a floating point number with sign $S \in \{0, 1\}$, exponent E and mantissa $M \in [1, 2)$ is then:

$$(-1)^S \cdot 2^E \cdot M.$$

Furthermore, the minimum relative error representable is called the *roundoff precision*, and is equal to 2^{-n_M-1}.

In Table 3.1 we report the details of the main IEEE 754 standards, that we use in this work.

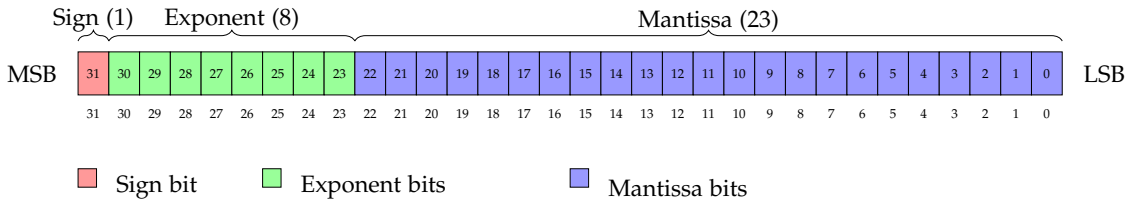

Figure 3.1: Scheme of IEEE 754 single precision floating point number representation.

[1] In the case the exponent has its minimum value, there is no implicit one and the mantissa is between 0 and 1. This is the denormalized case. Other special cases (infinity, not-a-number) happen when exponent is at one of its extreme values and mantissa is all 0 or all 1. For simplicity in this work we always assume values are normalized and finite.

Table 3.1: IEEE 754 floating-point formats: bit allocation (single sign bit omitted) and roundoff precision.

Name	Format	Exponent (bits)	Mantissa (bits)	Roundoff precision
half precision	$\mathbb{F}_{16}$	5	10	$u_h = 2^{-11}$
single precision	$\mathbb{F}_{32}$	8	23	$u_s = 2^{-24}$
double precision	$\mathbb{F}_{64}$	11	52	$u_d = 2^{-53}$

Alternative standards In recent years, other non-standard floating point data types have been proposed. Such alternative standards have gained popularity in recent years, due to the needs of Large Language Model (LLM) inference and training. They still follow the same scheme described above, composed of sign, exponent and mantissa, but differ in the number of bits assigned to each. Among these, `bfloat16` is commonly used in machine learning applications, as it offers range equal to $\mathbb{F}_{16}$ while reducing the mantissa to fit in 16 bits. Also OCP-FP8 (FP8-E4M3) is noteworthy, as it has been successfully used to emulate double-precision computation faster than native $\mathbb{F}_{64}$ on Graphics Processing Units (GPUs) [41].

3.1.2 Hardware Architecture

TPUs, NPUs and GPUs

The main computational task present in AI workloads is general matrix multiply (GEMM). The TPU [22] and NPU [25] are Application Specific Integrated Circuits (ASICs) designed for this task, but they evolved into a platform with the scope of accelerating the whole LLM inference task by integrating also fast vector operations. GPUs on the other hand were first designed for computer graphics, but were found to be useful also for AI workloads. The GPU is more flexible than the first two, as it contains a large number of small independent general-purpose cores. Nowadays GPUs contain also Tensor Cores specialized in GEMM.

Another notable feature that differentiates GPUs from the first two is that they often support double precision types, as part of their traditional general-purpose function. This can make the implementation of mixed-precision schemes easier, possibly with an impact on their performance. Nevertheless, in recent years the performance of native double precision operations has increased at a significantly slower rate than low-precision types.

A common feature in all three designs is the use of High Bandwidth Memory (HBM). Given the ample parallelism present in GEMM and vector operations, to have good utilization of the compute units it is important to have a high-throughput memory that would otherwise quickly become the bottleneck. Nevertheless, many applications remain memory-bound, this is the case for irregular memory access patterns, such as stencil, and sparse linear algebra computations.

The similarity of these architectures is apparent in the theoretical work on the TCU model, which develops a common theoretical framework for all these architectures [8].

Ascend 910B architecture

In this work we use Ascend NPUs to evaluate the proposed method. Therefore the DaVinci architecture of the Ascend 910B4 deserves some special attention.

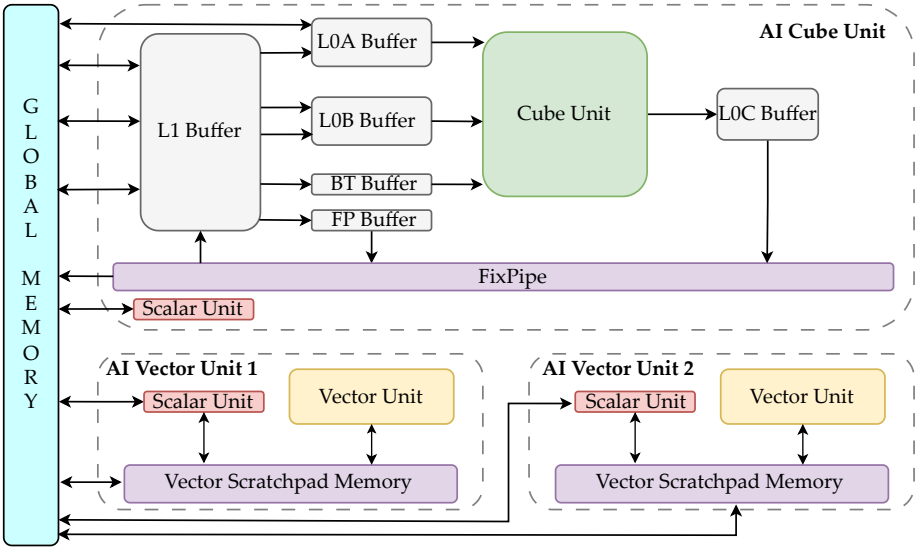

Figure 3.2: Scheme of a single AI Core of the 910B NPU [46].

The accelerator contains 20 AI cores [25, 46], each composed of one AI Cube (AIC) unit and two AI Vector (AIV) units. The architecture of each AI core is shown in Figure 3.2. Each unit contains a compute engine, a scalar unit and a Memory Transfer Engine (MTE). The compute engine is specialized for matrix multiplication in the case of AIC unit (Cube unit), and wide vector operations for AIV unit (Vector unit); the scalar unit is responsible for control flow and

scalar operations; the MTE handles memory operations independently of the other units.

The NPU contains a Global Memory (GM), shared between all AI cores. There is also a global L2 cache between GM and the compute cores. Each unit also has some private Scratchpad memory. The AIC unit has additionally some levels of buffers to maximize the utilization of the Cube unit. All memory movements are controlled in software, except for L2 caching.

3.1.3 Ozaki Schemes

Given two matrices $A \in \mathbb{F}_h^{p\times q}, B \in \mathbb{F}_h^{q\times r}$ in high precision $\mathbb{F}_h$ (usually double precision), the task that the Ozaki Scheme (OS) computes is a matrix multiplication operation AB.

The main advantage of the OS is performance: by using the fast Tensor Cores present in the hardware, the computation can be faster than native high-precision operations. Another advantage is reproducibility: since the scheme is implemented at a high level, with low-precision operations guaranteed to avoid overflows and rounding errors, the details of the rounding implementation in the hardware do not change the values, guaranteeing perfect reproducibility across platforms [32].

Let $\mathbb{F}_l$ be a low-precision type supported by hardware Tensor Cores. The scheme splits the matrices into low-precision slices:

$$A = A_1 + A_2 + \cdots + A_k, \qquad B = B_1 + B_2 + \cdots + B_\ell, \tag{3.1}$$

where

$$A_i \in \mathbb{F}_l^{p\times q}, \quad B_j \in \mathbb{F}_l^{q\times r}$$

for $1 \le i \le k$, $1 \le j \le \ell$ and for some lower precision $\mathbb{F}_l$ with $l < h$. Such slices are constructed iteratively as $A_i = \texttt{fl}\left(A - \sum_{j=0}^{i-1} A_j\right)$, where `fl` denotes the truncation to $\mathbb{F}_l$.

Then the desired product is approximated as the sum of pairwise products:

$$AB = (A_1 + A_2 + \cdots + A_k)(B_1 + B_2 + \cdots + B_\ell) = \sum_{i,j} A_i B_j. \tag{3.2}$$

This sum can be truncated: the size of some products $A_i B_j$ is smaller than the roundoff precision of $\mathbb{F}_h$ [40]. Also note that the scheme has to make sure that

the slices contain values small enough, so that the pairwise products $A_i B_j$ are guaranteed to be exact in the working precisions.

The first version of the OS [36] is also known as *mantissa slicing*, since the slices are constructed by splitting the mantissa of each entry in A and B.

Ozaki Scheme I using integer matrix multiplication

The scheme described above uses low-precision floating point operations on the slices. If not done carefully that step can easily produce overflow or inaccurate results. A solution for these problems is to scale the slices such that they contain small integer entries. In this regime, many operations between floating point numbers are exact and do not incur in under- and overflow. But since the values are small integers, the scheme can indeed use integer matrix multiplication [34]. Using integers can improve the memory efficiency of the scheme, as they avoid keeping auxiliary bits (exponent, sign). But ultimately the best option is usually dictated by hardware support and performance.

The scaling of the slices can be done in different ways. The more general form is the following: we consider a scaling factor for each row of A_i and column of B_j. Scaling factors are usually chosen as powers of two, so that they are guaranteed to be exact and they can be multiplied fast. Let such values be the entries of vectors $\mu \in \mathbb{Z}^n, \nu \in \mathbb{Z}^n$. Let also $\text{diag}(\mu^{(i)})$ and $\text{diag}(\nu^{(j)})$ be the diagonal matrices with the same values on the diagonal. Then the matrices are scaled as $\tilde{A}_i = \text{diag}(\mu^{(i)})A$ and $\tilde{B}_j = B_j \text{diag}(\nu^{(j)})$. Thus the final pairwise products are easy to reconstruct as:

$$A_i B_j = \text{diag}(\mu^{(i)})^{-1} \tilde{A}_i \tilde{B}_j \text{diag}(\nu^{(j)})^{-1}.$$

It is possible to use a scalar instead of a diagonal matrix on either side, but that may in turn require more slices to keep the same elementwise precision in the result.

Similar techniques are used in LLMs quantization [33], though in that case the goal is to minimize the memory footprint and numerical inaccuracy is acceptable.

Ozaki Scheme II

What is described until now, is known as OS I, since a new method to solve the same problem has been developed: the OS II, where the high-precision emulation is done using modular techniques and the Chinese Remainder Theorem [37].

The first step is to scale the input matrices A, B such that they have integer entries. This is done as explained in the previous section, thus obtaining integer matrices $\bar{A} = \text{diag}(\mu)\ A$ and $\bar{B} = B\ \text{diag}(\nu)$ After this step, the i-th slice $A^{(i)}$ is computed as:

$$A^{(i)}_{jk} = \bar{A}_{jk} \mod p_i,$$

where $p_1, \ldots, p_k \in \mathbb{N}$ are selected coprime integers. Similarly B is decomposed into $B_1, \ldots, B_k$ using the same moduli.

Then the products under each modulo are computed: $A_1B_1 \mod p_1, A_2B_2 \mod p_2, \ldots, A_kB_k \mod p_k$. On these the Chinese Remainder Theorem is applied on each entry to find the scaled result $\bar{A}\bar{B} \mod P$, with $P := \prod_{i=1}^{k} p_i$. Now if P is large enough and $P > 2|(\bar{A}\bar{B})_{i,j}|$ for all i, j, the value of $(\bar{A}\bar{B})_{i,j}$ modulo P is the same as in $\mathbb{Z}$.

At this point, we can finally compute the result AB by applying the inverse scaling, as:

$$AB = \text{diag}(\mu)^{-1}\ \bar{A}\bar{B}\ \text{diag}(\nu)^{-1}.$$

Of course the values $p_1, \ldots, p_k$ have to be chosen wisely: they have to guarantee that their product P is large enough, maximizing the number of computed digits in the result. But at the same time each slice has to fit in the low-precision type, limiting the size of the individual moduli p_i.

3.1.4 Iterative Refinement

The general scheme of Iterative Refinement (IR) works as described in Algorithm 5 [43]. It works by iterating three basic steps: residual reconstruction (Section 3.1.4), subsystem solution (Section 3.1.4) and solution update (Section 3.1.4).

Algorithm 5 General Iterative Refinement scheme.

Input: $A \in \mathbb{F}^{n\times n}$, $b, x_0 \in \mathbb{F}^n$,
$x \leftarrow x_0$
while not converged **do**
    $r \leftarrow b - Ax$
    $d \leftarrow \texttt{SOLVE}(A, r)$
    $x \leftarrow x + d$
end while
return x

This scheme enables the use of low-precision computation (which often can achieve more operations per second) for the most expensive step of the solution: the subsystem solution. In fact each of the three steps can be done with a different level of precision [7]. Though clearly the scheme can only converge to a solution up to the working precision of the update.

The algorithm used for subsystem solution step can be anything. The most popular schemes use LU decomposition, or GMRES.

3.1.5 Related Work

The idea of Iterative Refinement has been around for a long time, being first proposed by James H. Wilkinson in the 1960s [45]. Initially, IR was motivated by the performance gap between single and double precision. In more recent years it has seen a renaissance due to the widespread adoption of low-precision types in hardware. Therefore in the last years mixed-precision methods have been at the center of research [16, 1].

Specifically for solving linear systems, IR methods have been applied to dense problems on GPUs [16, 17]. This has been supported by the theoretical study of mixed-precision techniques to understand the conditions for their convergence [7]. The traditional limitation has been that the low-precision solution required $\kappa(A) < u^{-1}$, but more recent work has found techniques to extend their applicability [6].

On the other hand, for very similar reasons, Ozaki Schemes have recently gained popularity. So much so that they have been used to emulate double-precision GEMM with better performance than native operations [41].

Notation

This is a good place to introduce notation used in this section.

- Let $\mathbb{I}_b$ be the set of (signed) integers with b bits, so the set:$[-2^b, 2^b - 1] \cap \mathbb{Z}$.
- Let $\mathbb{F}_b$ be the set of floating point numbers with b bits. Refer to Table 3.1 for the specific numbers of bits and roundoff precision u of the formats. We denote with $\mathbb{F}_l$ and $\mathbb{F}_h$ a "low" precision and a "high" precision format respectively. We choose the first so that it is supported natively by the NPU hardware, but we don't require the same for the second, which may be arbitrarily accurate.

- We use L1, L2 and infinity norms, for vectors:

$$\|v\|_1 := \sum_i |v_i|, \qquad \|v\|_2 := \sqrt{\sum_i v_i^2}, \qquad \|v\|_\infty := \max_i |v_i|;$$

for matrices, in general, given a vector norm $\|\cdot\|$ the corresponding matrix norm is defined as:

$$\|A\| := \max_{v:\|v\|=1} \|Av\|.$$

But the selected norms can be expressed also in more useful terms, as:

$$\|A\|_1 := \max_j \sum_{i=1}^{n} |A_{ij}|, \qquad \|A\|_2 := \rho(A), \qquad \|A\|_\infty := \max_i \sum_{j=1}^{n} |A_{ij}|;$$

where $\rho(A)$ is the spectral radius of A i.e. the magnitude of largest eigenvalue.

Remark the following inequalities between the norms: for any $v \in \mathbb{R}^n$ it holds that

$$\|v\|_2 \leq \|v\|_1 \leq n\|v\|_\infty. \tag{3.3}$$

- We also use the max-norm of a matrix: $\|A\|_{\max} := \max_{i,j} |A_{ij}|$, note though that it doesn't have a corresponding vector norm.

- To quantify the conditioning of a matrix, we consider the L2 condition number: $\kappa(A) := \|A\|_2\|A^{-1}\|_2$.

3.2 Iterative Refinement Method

We are given a sparse matrix $A \in \mathbb{F}_h^{n\times n}$, and a vector $b \in \mathbb{F}_h^n$. The task is to solve the linear system $Ax = b$, finding $x \in \mathbb{F}_h^n$.

Intuition The development of the method started by considering an iterative method where A is divided into a sum of slices $A^{(0)} + A^{(1)} + \cdots + A^{(N_A-1)}$, where each is of a smaller "magnitude" than the previous ones (for some meaning of "magnitude"). Similarly we assume that the solution vector x is expressed as a sum $x^{(0)} + x^{(1)} + \cdots + x^{(N_x-1)}$. With this notation we have that:

$$Ax = A^{(0)}x^{(0)} + (A^{(1)}x^{(0)} + A^{(0)}x^{(1)}) + (A^{(2)}x^{(0)} + A^{(1)}x^{(1)} + A^{(0)}x^{(2)}) + \cdots$$

where the parentheses group products of similar "magnitude". The idea now is to iteratively find $x^{(i)}$ by removing all the known contributions of larger or similar magnitude, and solving the residual linear system:

$$r^{(i)} = b - A^{(0)}x^{(0)} - (A^{(1)}x^{(0)} + A^{(0)}x^{(1)}) - \cdots - \left(\sum_{j=1}^{i} A^{(j)}x^{(i-j)}\right) = A^{(0)}x^{(i)}.$$

In the end this same method can be seen as a standard iterative refinement that uses an Ozaki scheme with adaptive precision in the approximation of the residual. Further analysis made clear that the precision in the computation of the residual should be kept constant, to increase the quality of the next $x^{(i)}$. Thus we obtained the method presented next.

3.2.1 Ozaki Iterative Refinement

The initial step of the algorithm is to split $A = A^{(0)} + A^{(1)} + \cdots + A^{(N_A-1)}$ with $A^{(0)}, A^{(1)}, \cdots, A^{(N_A-1)} \in \mathbb{F}_l$. This is done exactly as in the Ozaki Scheme I, but possibly with fewer significant bits per slice. Let also $\hat{A} \in \mathbb{F}_\ell$ be the such that it minimizes $\|A - \hat{A}\|_{\max}$. This is useful, since the second might have further constraints, as we'll see later.

In the algorithm we call a subroutine to solve a linear system $Mx = b$ in low-precision $\mathbb{F}_l$. We denote such procedure as $\texttt{SOLVE}(M, b, \varepsilon_S)$, where ε_S is the target relative forward error i.e. the output $x \in \mathbb{F}_l^n$ is s.t. $\|Mx - b\|_2 \leq \varepsilon_S\|b\|$. This condition only makes sense for sufficiently big $\varepsilon_S \in (0, 1)$ and M well conditioned, since the subroutine runs in low-precision. In principle SOLVE can be

any algorithm. Given our focus on large sparse matrices, we implemented SOLVE with a Conjugate Gradient (CG). Thus we must add the additional constraint that A be Symmetric Positive Definite (SPD). Nevertheless, this additional condition is otherwise not needed for the IR method to work.

The proposed algorithm is described in Algorithm 6. This is a standard iterative refinement scheme, but the interesting addition is that the residual estimate is refined with an Ozaki-like scheme.

Algorithm 6 Ozaki-based Iterative Refinement solver.

Input: $A^{(0)}, A^{(1)}, \cdots, A^{(N_A-1)}, \hat{A} \in \mathbb{F}_\ell^{n\times n}$, $b \in \mathbb{F}_h^n$, $\varepsilon_O \in \mathbb{R}_{>0}$
$k = 0$
$r^{(0)} \leftarrow b$
$\alpha \leftarrow \frac{\varepsilon_O}{1+kN_A}$
while $\|r^{(k)}\|_2 > \varepsilon_O$ **do**
    $k \leftarrow k+1$
    $r^{(k)} \leftarrow r^{(k-1)}$
    $x^{(k)} \leftarrow \texttt{SOLVE}(\hat{A}, r^{(k)}, \varepsilon_S)$
    for $j = 0$ to $N_A - 1$ **do**
        if $\|A^{(j)}\|_1 \|x^{(k)}\|_1 \geq \alpha$ **then**
            $r^{(k)} \leftarrow r^{(k)} - A^{(j)}x^{(k)}$
        end if
    end for
end while
return $\sum_{i=0}^{k} x^{(i)}$

Scaling We represent $A^{(0)}, A^{(1)}, \cdots, A^{(N_A-1)} \in \mathbb{F}_l^{n\times n}$ using a single scalar scaling factor for the whole matrix i.e. $A^{(i)} = 2^{a_i}\tilde{A}^{(i)}$ for some $\tilde{A}^{(i)} \in \mathbb{Z}^{n\times n}$. This is different from what is usually done in an Ozaki Scheme, that uses a right-hand side multiplication by a diagonal scaling matrix. This simpler scaling is allowed by the different precision requirement. High-precision matrix multiplication emulation means aiming for elementwise precision, while for a linear solver the standard error metric is L2 norm (or possibly infinity norm i.e. the maximum elementwise error). Using a scalar scaling factor might result in large componentwise relative errors for small elements, but the large elements dominate these

global metrics. We prove in Section 3.2.3 that for L2 norm the method is well-behaved.

Note that a row-wise/column-wise scaling of the matrix, can be seen as a form of preconditioning, so it may be an effective way to extend the applicability of the method. Preconditioning is not considered in this work: we therefore limit to well-conditioned matrices.

Splitting and reconstructing The splitting of matrix A into the sum $A^{(0)} + A^{(1)} + \cdots + A^{(N_A-1)}$ and the final sum $\sum_{i=0}^{k} x^{(i)}$ are naturally implemented using operations in $\mathbb{F}_h$. These operations can technically be done on Ascend NPUs, but the implementation would be complex since the hardware has no native support for 64-bit data types. Therefore the operations would have to be implemented by decomposing the target floating point value into mantissa and exponent, possibly dividing again those values into 32-bit parts, in software.

Therefore, for ease of implementation, we compute these steps on the CPU. This step takes a time on the same order as the execution of the IR, suggesting that its acceleration using NPU could speed up the end-to-end runtime of the scheme, from high-precision inputs to high-precision solution. On different hardware, such as many GPUs, 64-bit types are natively supported, and this step can be easily implemented on the device too.

We observed that the time to sum and move all $x^{(i)}$ from the device to the CPU was only a small fraction of the total. This step takes from $\sim 2 \cdot 10^{-4}$ (for $n = 2^{11}$) to $\sim 3 \cdot 10^{-3}$ seconds (for $n = 2^{16}$), always more than one order of magnitude less than the overall runtime. Also a simple asymptotic consideration supports the observations: the total runtime is $\mathcal{O}(\#\text{nonzeros in } A)$, while the final sum takes only $\mathcal{O}(n)$ time. Therefore this does not seem a limitation of this implementation or the hardware in this context.

Representation of the residual Since the residual has to be kept with high precision, in the implementation we use a fixed-digit representation using multiple $\mathbb{I}_{32}$ per element together with a single scalar scaling factor. In this way we can keep an accurate residual using only $\mathbb{I}_{32}$ operations.

It is also possible, and simpler in some cases, to implement this reconstruction using floating point numbers (e.g. $\mathbb{F}_{32}$) and without handling additional scaling factors. Nevertheless using integers uses a smaller memory footprint and thus smaller memory movements.

Choice of norms The choice of using L1 norm in the inner loop is mostly for ease of computation: computing the L2 norm of a matrix is, in general, very expensive. On the other hand, computing the L1 norm only requires simple sums: $\|x\|_1 = \sum_i |x^{(i)}|$ and $\|A^{(k)}\|_1 = 2^{a_k} \max_j \sum_{i=1}^{n} \left|\tilde{A}_{ij}^{(k)}\right|$. In the implementation, we compute this second value offline. We validate this choice in Section 3.2.3. Also infinity norm would work, but would give worse L2 precision guarantees, as remarked in Equation (3.3).

3.2.2 Considerations on the SpMV

We have Sparse Matrix–Vector products (SpMVs) inside the `SOLVE` procedure and in the residual refinement inner loop. The implementation of these on the NPU is not trivial, and this is an active field of research. We compute these using an algorithm developed by Soboczyk et al. [39]. We describe and analyze the algorithm in this section.

The method takes as inputs a sparse matrix $A \in K^{n \times n}$ in Compressed Row Storage (CSR) format and a vector $x \in K^n$. The CSR format represents the matrix by storing three vectors: $A.val \in K^{n_{nz}}$, containing nonzero values, $A.col \in \mathbb{N}^{n_{nz}}$ containing column indices and $A.row \in \mathbb{N}^{n+1}$ indicating the endpoints of each row's entries.

To compute $y = Ax$ it proceeds in three steps:

1. *Gather-product*: compute $z[i] = A.val[i] \cdot x[A.col[i]]$ (for each nonzero entry in A);
2. *Scan*: compute $s[i] = \sum_{j=0}^{i} z[j]$. This is the only step that is accelerated using the Cube unit, in addition to the vector unit.
3. *Gather-diff*: compute $y_i = s[A.row[i+1]] - s[A.row[i]]$.

Future refinements of the method aim to fuse the last two steps, or simply partition the sum. This will improve the numerical stability by avoiding the computation of large sums and subtractions between them, which currently limits the applicability of the method.

Data types Supported input data types are $\mathbb{F}_{32}$, $\mathbb{F}_{16}$ and $\mathbb{I}_{16}$, as these are the only supported input types for vector `mul` operation. In the last case z needs to be truncated to $\mathbb{I}_8$ to use Cube acceleration for the scan.

The vector s has entries of type $\mathbb{F}_{32}$ in both floating point cases, and $\mathbb{I}_{32}$ in the integer case. The final result y has the same type as s.

Limitations In the design of the IR method, specifically in the reconstruction of the residual, it is essential to have a good understanding of the SpMV's precision. Therefore let us analyze some sufficient conditions under which the computation is exact.

Let's call ℓ and h respectively the number of significant bits allowed as input of the *Scan*, and as final output. To have an exact result:

- in the *Gather-product* step: the entries in z cannot overflow; so the number of significant bits in the input shall be at most $\lfloor \ell/2 \rfloor$;
- in the *Scan* step: s has h significant bits, while z has ℓ significant bits. If the maximum number of elements in the sum is at most $2^{h-\ell}$.

  Subsequently, the *Gather-diff* step makes a single subtraction between two h-bit numbers, requiring at most one more significant bit. Therefore the maximum number of nonzeros has to be limited to $2^{h-\ell-1}$.

With these conditions, we can compute the maximum number of significant bits that we can allow in the input, and maximum number of nonzeros so that at each step the computations are correct. These resulting numerical values are summarized in Table 3.2 for the different possible input types.

	Input type		
	$\mathbb{I}_{16}$	$\mathbb{F}_{16}$	$\mathbb{F}_{32}$
Scan capacity ℓ	7	11	23
Output capacity h	31	23	23
Max number of bits in input $\lfloor \ell/2 \rfloor$	3	5	11
Max number of nonzeros $2^{h-\ell-1}$	$2^{31-7-1} = 2^{23} \sim 8 \cdot 10^6$	$2^{11} = 2048$	1

Table 3.2: Summary of number of significant bits at each step of the SpMV.

From these results it is clear that both $\mathbb{F}_{16}$ and $\mathbb{F}_{32}$ are impractical given the small number of nonzeros they can handle without error, though of course in real cases the number is almost always larger than the given worst-case estimate. This therefore motivates the choice of using $\mathbb{I}_{16}$ in the residual reconstruction.

The use of an integer type for this SpMV is not limiting, since it is sufficient to scale the slices $A^{(j)}$ and $x^{(i)}$ to keep the desired number of significant bits.

It is possible to further refine the method by allowing the number of significant bits in the right-hand side and left-hand side to be different, and thus make use of one additional bit of precision. In the final version of the method we use $s_A = 3$ significant bits in each $A^{(i)}$ and $s_x = 4$ in each $x^{(j)}$. To guarantee s_x correct bits in $x^{(i)}$ implies that $\varepsilon_S \lesssim 2^{-s_x}$.

`SOLVE` **SpMV precision** The `SOLVE` procedure will, for virtually any algorithm, make use of SpMV internally. Therefore it makes sense to briefly consider the tradeoffs in the use of each data type available in this context. First, using integer SpMV can be a valid option, but it may also imply limited precision in the representation of $\hat{A}$. Clearly, a scaling scheme can be used to keep high fidelity in the representation. Nevertheless the limitations in the number of significant bits in the entries are the same as discussed before (an overflow can be disastrous, as it may be undefined behavior), with only up to 3-4 usable bits.

For floating point types, the situation is different from before, as we no longer require an exact result and so can afford losing some low-value bits. In this way the precision degradation is gradual when increasing significant bits in the input. Therefore we only have to guard against over- and underflows. For this it is sufficient to scale the inputs such that the largest entries are e.g. between 1 and 2.

The choice between $\mathbb{F}_{16}$ and $\mathbb{F}_{32}$ is probably not very important: in both cases the output is single precision, and so are all the vector operations. The accuracy of the result is similar, as both use single precision in the sums (*Scan* and *Gather-diff* steps) and the multiplications (*Gather-diff* step) are stable operations. Because of higher bandwidth used by $\mathbb{F}_{32}$, its performance is slightly worse than half-precision. Therefore we'll use $\mathbb{F}_{16}$.

Splitting parameters setting Having fixed the number of significant bits in each $A^{(i)}$ and $x^{(j)}$, this determines the precision that a number of parts N_A and k gives:

$$\left\| A - \sum_{i=0}^{N_A-1} A^{(i)} \right\|_{\max} \leq 2^{-s_A \cdot N_A}.$$

Therefore given ε_O we can set $N_A = \lceil -\log_2(\varepsilon_O/(2n))/s_A \rceil$ to have that:

$$\|\varepsilon_A\|_2 := \left\| A - \sum_{i=0}^{N_A-1} A^{(i)} \right\|_2 \leq n \left\| A - \sum_{i=0}^{N_A-1} A^{(i)} \right\|_{\max} \leq n2^{-s_A \cdot N_A} \leq \varepsilon_O.$$

3.2.3 Convergence Analysis

Throughout this section we call:

- $\overline{x}^{(i)} := \sum_{j=0}^{i} x^{(j)}$, i.e. the approximation of the solution after iteration i;
- $\overline{r}^{(i)} := b - A\overline{x}^{(i)}$, i.e. the true residual of $\overline{x}^{(i)}$;
- $r^{(i)}$ the estimated residual at the end of iteration i;
- k the **total** number of iterations of the algorithm;
- u the roundoff precision of $\mathbb{F}_l$.

Furthermore, assume that $\|A\|_2 = \mathcal{O}(1)$ and $\|b\|_2 = \mathcal{O}(1)$. This assumption doesn't lose generality, since it is sufficient to scale the inputs to satisfy this assumption. Also assume that $\kappa(A)$ is constant. This is a simplifying assumption, but it does not decrease the value of the theorems, since they prove correctness only with bounded condition number.

In the algorithm, the residual is updated only counting contributions of at least a fixed size in L1 norm. Therefore we need a first result limiting the error in L2 norm of the residual.

Theorem 3.1 (Residual correctness)**.** *If the residual is stored with enough precision, at each iteration Algorithm 6 keeps an estimate of the residual $r^{(i)}$ such that*

$$\|r^{(i)} - \overline{r}^{(i)}\|_2 \leq \mathcal{O}(\varepsilon_O).$$

Proof. Let $\mathbb{1}_{i,j}$ be the following indicator function:

$$\mathbb{1}_{i,j} := \begin{cases} 1 & \text{if } \|A^{(j)}\|_1 \|x^{(i)}\|_1 \leq \alpha := \frac{\varepsilon_O}{1+kN_A} \\ 0 & \text{otherwise.} \end{cases}$$

Then, since $\|A^{(j)}x^{(i)}\|_1 \leq \|A^{(j)}\|_1\|x^{(i)}\|_1$, we have:

$$\sum_{m=0}^{i}\sum_{j=0}^{N_A-1}\mathbb{1}_{m,j}\|A^{(j)}x^{(m)}\|_2 \leq \sum_{m=0}^{i}\sum_{j=0}^{N_A-1}\mathbb{1}_{m,j}\|A^{(j)}x^{(m)}\|_1 \leq N_A k\alpha \leq \varepsilon_O.$$

Now, the residual can be expressed as:

$$r^{(i)} = b - \sum_{m=0}^{i}\sum_{j=0}^{N_A-1}(1-\mathbb{1}_{m,j})A^{(j)}x^{(m)} = \underbrace{b - A\overline{x}^{(i)}}_{\overline{r}^{(i)}} + \sum_{m=0}^{i}\sum_{j=0}^{N_A-1}\mathbb{1}_{m,j}A^{(j)}x^{(m)} + \varepsilon_A\overline{x}^{(i)}.$$

So a simple application of the triangle inequality yields:

$$\begin{aligned}\|r^{(i)} - \overline{r}^{(i)}\|_2 &\leq \left\|\sum_{m=0}^{i}\sum_{j=0}^{N_A-1}\mathbb{1}_{m,j}A^{(j)}x^{(m)} + \varepsilon_A\overline{x}^{(i)}\right\|_2 \\ &\leq \sum_{m=0}^{i}\sum_{j=0}^{N_A-1}\mathbb{1}_{m,j}\left\|A^{(j)}x^{(m)}\right\|_2 + \|\varepsilon_A\overline{x}^{(i)}\|_2 \leq \varepsilon_O + \|\varepsilon_A\overline{x}^{(i)}\|_2.\end{aligned}$$

Lastly we can conclude by noticing that:

$$\|\varepsilon_A\overline{x}^{(i)}\|_2 \leq \|\varepsilon_A\|_2\left\|A^{-1}\right\|_2\|b\|_2 \leq \|\varepsilon_A\|_2\frac{\kappa(A)}{\|A\|_2}\|b\|_2 \leq \mathcal{O}(\varepsilon_O). \qquad \square$$

Using this result, we can prove that the whole method converges to the desired precision.

Theorem 3.2 (Convergence correctness)**.** *If* $C := (1+u)u\kappa(\hat{A}) + \varepsilon_S < \frac{1}{2}$*, then* $\exists\overline{k} = \mathcal{O}\left(\log(\|b\|_2/\varepsilon_O)\right)$ *s.t. after* $\overline{k}$ *iterations, the backward error converges to the desired precision* ε_O*, that is:*

$$\|\overline{r}^{(\overline{k})}\|_2 = \|A\overline{x}^{(\overline{k})} - b\|_2 < \mathcal{O}(\varepsilon_O).$$

Furthermore the constants inside the big-$\mathcal{O}$ *notation only depend on* C*.*

Proof. By construction we have that: $\|A - \hat{A}\|_{\max} < u\|A\|_{\max}$. Hence:

$$\|A - \hat{A}\|_\infty < u\|A\|_{\max}.$$

Also observe that $\|A\|_{\max} \leq (1+u)\|\hat{A}\|_{\max} \leq (1+u)\|\hat{A}\|_2$ because entries are rounded to $\mathbb{F}_l$.

Note that at any iteration i, we have that $\overline{r}^{(i)} = \overline{r}^{(i-1)} - Ax^{(i)}$. Therefore, using Theorem 3.1:

$$\begin{aligned}
\|\overline{r}^{(i-1)} - Ax^{(i)}\|_2 &\leq \|Ax^{(i)} - \hat{A}x^{(i)}\|_2 + \|\hat{A}x^{(i)} - \overline{r}^{(i-1)}\|_2 + \|\overline{r}^{(i-1)} - r^{(i-1)}\|_2 \\
&\leq \|A - \hat{A}\|_2 \|x^{(i)}\|_2 + \varepsilon_S \|r^{(i-1)}\|_2 + \mathcal{O}(\varepsilon_O) \\
&\leq (1+u)u\|\hat{A}\|_2 \left\|\hat{A}^{-1}\right\|_2 \|r^{(i-1)}\|_2 + \varepsilon_S \|r^{(i-1)}\|_2 + \mathcal{O}(\varepsilon_O) \\
&\leq \left(\underbrace{(1+u)u\,\kappa(\hat{A}) + \varepsilon_S}_{C} \right) \|r^{(i-1)}\|_2 + \mathcal{O}(\varepsilon_O) \\
&\leq C\|\overline{r}^{(i-1)}\|_2 + \mathcal{O}(\varepsilon_O) \leq 2\max\{C\|\overline{r}^{(i-1)}\|_2, \mathcal{O}(\varepsilon_O)\}.
\end{aligned}$$

Now with a simple induction we can see that:

$$\begin{aligned}
\|\overline{r}^{(i)}\|_2 = \|\overline{r}^{(i-1)} - Ax^{(i)}\|_2 &\leq \max\{(2C)\|\overline{r}^{(i-1)}\|_2, \mathcal{O}(\varepsilon_O)\} \\
&\leq \max\{(2C)^i \|\overline{r}^{(0)}\|_2, \mathcal{O}(\varepsilon_O)\}.
\end{aligned}$$

Lastly note that if $\overline{k} = \mathcal{O}\left(\log(\|b\|_2/\varepsilon_O)\right)$, then $(2C)^{\overline{k}} = \mathcal{O}(\varepsilon_O/\|b\|_2)$ since $2C < 1$. Thus we can conclude by noting that the right hand side is a max between $\mathcal{O}(\varepsilon_O)$. □

3.2.4 Performance analysis

Let A be $n \times n$ with n_{nz} non-zero entries. Let also N_{res} be the number of parts in which the residual is split.

Memory movement analysis For simplicity we'll consider a single iteration and then multiply the result by the number of iterations. Thus we are ignoring possible reuse of variables between the iterations. The aim of this analysis is more to understand the fundamental limits of the IR method, not evaluate the quality of the implementation using details of the current SpMV or the chosen `SOLVE` procedure.

We'll lower-bound the bandwidth needed by `SOLVE` with the amount of data moved by a single SpMV: $((2+4)n_{nz} + (4+2)n)$ bytes. This value comes from considering `int32` column and row indices, and 2-byte input and output values (so $\mathbb{F}_{16}$ or $\mathbb{I}_{16}$).

Of course, this is a rough lower-bound and in practice it is quite likely that the data is accessed multiple times, especially with iterative solvers. For the residual reconstruction, we can count the exact number of SpMVs plus the number of vector operations done on the residual, for a total of $N_A(6n_{nz} + (4N_{res} + 6)n)$ bytes. The reconstruction of the residual in $\mathbb{F}_{16}$ and the computation of vector norms imply $\mathcal{O}(n)$ additional movements. We will ignore these additional movements, as they are asymptotically small compared to the SpMVs, which dominate the bandwidth required with $\mathcal{O}(n_{nz})$ bytes moved.

Therefore, we can summarize the overall bandwidth lower bound across k iterations as:

$$k\left(6n_{nz} + 6n + N_A(6n_{nz} + (4N_{res} + 12)n)\right) \text{ bytes.} \tag{3.4}$$

3.3 Experimental results

3.3.1 Test problems

Random problem generation As test problems we created a set of sparse SPD random matrices with controlled condition number and density. We start with a sparse random upper triangular matrix U with zero diagonal and nonzero entries independently sampled from a standard normal distribution. Then we consider the symmetric matrix $R = U + U^T$, and compute its maximum and minimum (with the canonical ordering of real numbers) eigenvalues $\lambda_{\max}$ and $\lambda_{\min}$. This makes sense, since R is real symmetric and therefore its eigenvalues are real.

Finally we can compute the desired SPD matrix A as

$$A = I + \frac{\kappa - 1}{\lambda_{\max} - \lambda_{\min}}(R - \lambda_{\min} I),$$

where I is the identity matrix of suitable dimension. Adding identity and scaling the matrix shifts and scales the spectrum, so A has extreme eigenvalues 1 and κ. Thus it is SPD and with $\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}} = \kappa$ as desired.

This construction has the added benefit that the nonzero structure of the matrix is independent of the desired condition number. This should allow the separate observation of the influence that the two properties have on the runtime.

The solution vector x is generated by independently sampling each entry $x_i \in \mathbb{F}_{64}$ uniformly from $[0, 1]$. Finally the vector b is found as Ax, computed in double precision.

Parameter space The parameters (n, d, κ), dimension, density and condition number are chosen from a grid. We consider all tuples $(n, d, \kappa) \in \mathcal{N} \times \mathcal{D} \times \mathcal{K}$, where:

$$\mathcal{N} = \{2^x | x \in \{11, 12, 13, 14, 15, 16\}\} = \{2048, 4096, 8192, 16384, 32768, 65536\},$$
$$\mathcal{D} = \{0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3\},$$
$$\mathcal{K} = \{2^x | x \in \{1, 2, \ldots, 12\}\} = \{4, 8, 16, 32, 64, 128, 256, 512, 1024\}.$$

These values are chosen such that they are approximately exponentially spaced in each variable, and span a range that approximately captures all reasonable values for each parameter. The parameter space is further limited to instances with less than approximately 10^8 nonzeros and with $n \leq 2^{16}$, because of limitations in the implementation the SpMV. Note that this limit on the number of nonzeros is larger than the (pessimistic) theoretical limit reported in Table 3.2.

3.3.2 Experimental setup

Baseline solvers We compare the proposed implementation of the IR method running on NPU against the multithreaded CPU implementations of CG from ALP/GraphBLAS [48, 30] and Eigen [15]. For the first, we use the `nonblocking` backend for two main reasons: first, this backend has been observed to be often faster than the multithreaded blocking; second, it implements an automatic tiling mechanism, ensuring good performance without manual parameter tuning.

We measure the runtime of the solvers at a fixed relative residual L2 norm target, equal to $\varepsilon_O = \frac{\|\bar{r}^{(i)}\|_2}{\|b\|_2} = 10^{-14}$.

Systems The IR runs on an Ascend 910B4 with CANN 9.0.0, with host CPU an AMD Epyc 9654. We will call this **NPU system**.

The CPU baselines run in double precision on the following systems:

- **ARM**: HiSilicon Kunpeng-920 CPU (2 sockets, 48 cores per socket), 512 GiB RAM.
- **Cascade**: Intel Xeon Gold 6238T CPU (2 sockets, 22 cores per socket), 180 GiB RAM.

All systems run Ubuntu 22.04; The compiler used is GCC 11.4 with flags `-O2 -march=native -mtune=native`. The kernel version is 5.15.0 for the first two, and

6.8.0 for the NPU system. We run the CPU baselines with the following OpenMP settings: `OMP_PLACES=cores` and `OMP_PROC_BIND=close`. The number of threads is set to the number of physical cores, so 44 for Cascade and 96 for ARM.

Furthermore, the runtimes will be compared also with the memory movement time estimate using Equation (3.4) and the measured peak bandwidth value of $1.47\,\mathrm{TB\,s^{-1}}$ for GM.

3.3.3 Results

The measured runtimes and residuals for a selection of condition numbers is shown in Figure 3.3. Some data points are missing: for large nonzero count and high condition number, the construction makes the entries larger, and thus the SpMV less numerically unstable. Above a certain threshold the method no longer converges, and thus no data could be gathered.

From these, we can observe a few important things: first, the nonzero count is indeed the dominant factor in determining the runtime. Density and matrix dimension seem to have only a minor impact. Second, the runtime of IR is slower than the baselines for small matrices, while being comparable for larger ones. This is definitely caused by the overhead of communication between the CPU and the NPU. Third, while the residual is quite similar for the two CG baselines, IR shows the error metrics are almost always smaller than CG. This is likely due to the fact that the inner solve always solves with the same fixed precision, thus each outer iteration produces larger steps compared to CG. Nevertheless this does not explain the larger gap between forward errors compared to residual errors. It is possible that the residual reconstruction scheme is more accurate than the double precision residual kept by CG, thus making the final solution more precise.

To better understand how the runtimes of the different solvers compare, consider the speedup of the baselines against IR (i.e. the baseline runtime divided by IR runtime). These values are reported in the last row of Figure 3.3. The figure clearly shows that IR is faster than Eigen on large instances and for a range of condition numbers, while consistently being slower than ALP/GraphBLAS' `nonblocking` baseline.

Nevertheless, IR's runtime has the same order of magnitude as the baselines. The runtime lower bound given by the roofline model is smaller than all baselines, but still more than 20 times smaller than measured runtimes. This would

suggest that the implementation is quite inefficient. But the roofline estimate only considers bandwidth and ignores latency and synchronization costs. Furthermore some memory movements are deliberately excluded, such as the inner workings of SpMV.

Furthermore, the plots show the runtime lower bounds given by the roofline estimate from Equation (3.4). These lower bounds are consistently one order of magnitude smaller than the best measured runtime. This shows that the algorithms for SpMV and IR have still room to improve. In particular, together with improvements in NPU hardware, IR methods could make NPU acceleration of sparse workloads a competitive option compared to established CPU solvers.

3.4 Conclusion

In this chapter we have explored a novel Iterative Refinement method, aimed at using NPUs. We analyzed, implemented and evaluated the algorithm, observing the limitations of the hardware and approaches to work around them. This method, based on the OS I, shows performance competitive with established sparse solvers.

The proposed method can approximate arbitrary precision, though its complexity increases quadratically: both N_A and k increase linearly. To have a linear complexity, an idea would be to adapting the OS II. But doing solving linear systems using modular techniques in the same way as was done with OS I is probably harder if not impossible. The IR method works on the real numbers by iteratively solving a low-precision version of the system. This approach works on the real numbers, where the euclidean space has the same structure independently of the scale, but modular spaces are fundamentally different. So a modular technique similar to OS II needs a different approach.

Mixed precision approaches, and especially emulation schemes, show a possible paradigm shift happening in Scientific Computing. When using these techniques, precision becomes a parameter that can be changed, tuned and optimized. This is in stark contrast with the traditional point of view, where the working precision is determined by the hardware's features.

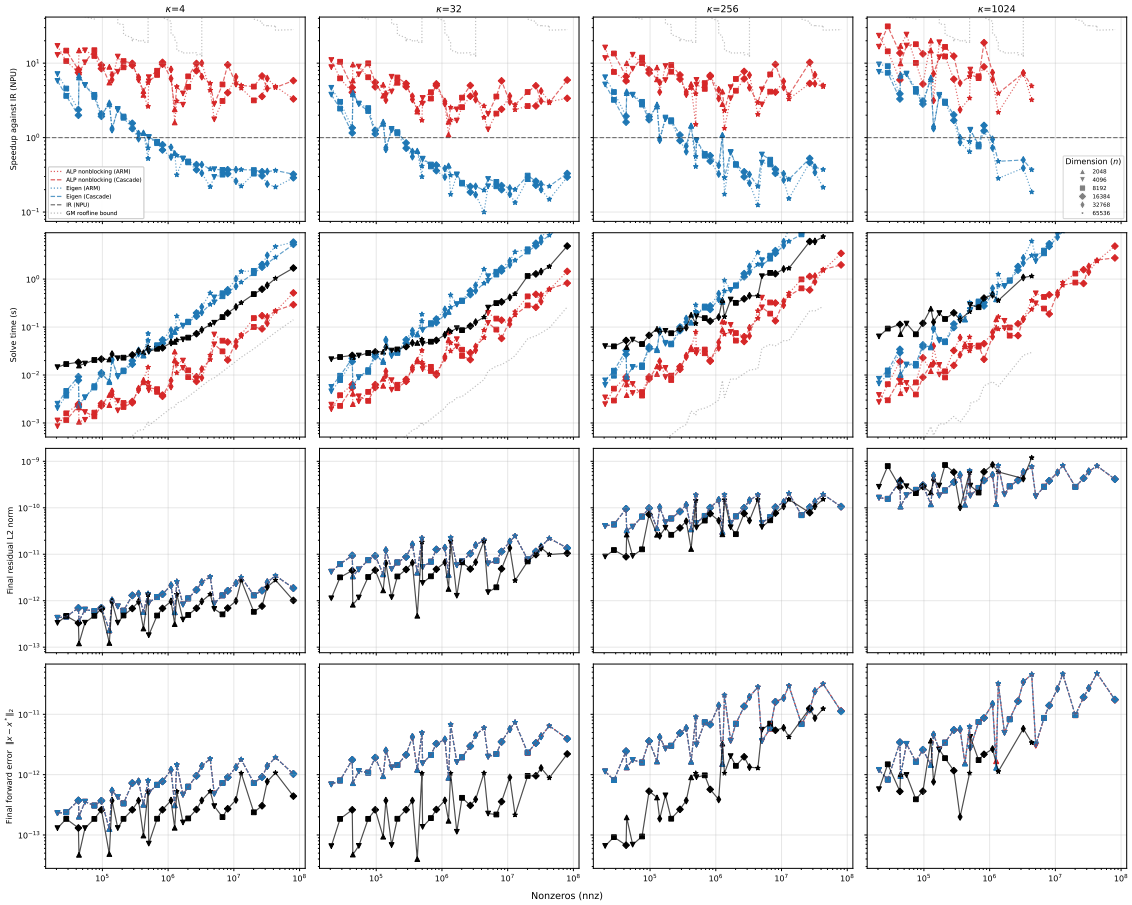

Figure 3.3: Each row of plots compares a different value on the y axis against matrix nonzero count for the different solvers. Starting from the top, the values on the y axis are: speedup (runtime /IR runtime), runtime, residual and forward error (L2 norm). Condition number κ is constant on each column.

Bibliography

[1] Ahmad Abdelfattah, Hartwig Anzt, Erik G Boman, Erin Carson, Terry Cojean, Jack Dongarra, Alyson Fox, Mark Gates, Nicholas J Higham, Xiaoye S Li, Jennifer Loe, Piotr Luszczek, Srikara Pranesh, Siva Rajamanickam, Tobias Ribizel, Barry F Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung M Tsai, and Ulrike Meier Yang. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *Int. J. High Perform. Comput. Appl.*, 35(4):344–369, July 2021. ISSN 1094-3420. doi: 10.1177/10943420211003313. URL `https://doi.org/10.1177/10943420211003313`.

[2] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 145–158, 2020. doi: 10.1109/ISCA45697.2020.00023.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring)*, 1967. URL `https://api.semanticscholar.org/CorpusID:195607370`.

[4] Maliheh Aramon, Gili Rosenberg, Elisabetta Valiante, Toshiyuki Miyazawa, Hirotaka Tamura, and Helmut G Katzgraber. Physics-inspired optimization

for quadratic unconstrained problems using a digital annealer. *Frontiers in Physics*, 7:48, 2019.

[5] Benjamin Brock, Aydın Buluç, Raye Kimmerer, Jim Kitchen, Manoj Kumar, Timothy Mattson, Scott McMillan, José Moreira, Michel Pelletier, and Erik Welch. Graphblas c api specification, 2023. URL `https://graphblas.org/docs/GraphBLAS_API_C_v2.1.0.pdf`.

[6] Erin Carson and Nicholas J. Higham. A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM Journal on Scientific Computing*, 39(6):A2834–A2856, 2017. doi: 10.1137/17M1122918. URL `https://doi.org/10.1137/17M1122918`.

[7] Erin Carson and Nicholas J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing*, 40(2):A817–A847, 2018. doi: 10.1137/17M1140819. URL `https://doi.org/10.1137/17M1140819`.

[8] Rezaul Chowdhury, Francesco Silvestri, and Flavio Vella. A computational model for tensor core units, 2020. URL `https://arxiv.org/abs/1908.06649`.

[9] Agnieszka Debudaj-Grabysz and Rolf Rabenseifner. Nesting openmp in mpi to implement a hybrid communication method of parallel simulated annealing on a cluster of smp nodes. In Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 18–27, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31943-6.

[10] Iain Dunning, Swati Gupta, and John Silberholz. What works best when? a systematic evaluation of heuristics for max-cut and QUBO. *INFORMS Journal on Computing*, 30(3), 2018.

[11] David J. Earl and Michael W. Deem. Parallel tempering: Theory, applications, and new perspectives. *Physical Chemistry Chemical Physics*, 7(23):3910, 2005. ISSN 1463-9076, 1463-9084. doi: 10.1039/b509983h.

[12] Ana M. Ferreiro, José Antonio García-Rodríguez, José G. López-Salas, and Carlos Vázquez. An efficient implementation of parallel simulated anneal-

ing algorithm in gpus. *CoRR*, abs/2408.00018, 2024. doi: 10.48550/ARXIV.2408.00018. URL https://doi.org/10.48550/arXiv.2408.00018.

[13] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024. doi: 10.1109/MM.2024.3373763.

[14] Hayato Goto, Kotaro Endo, Masaru Suzuki, Yoshisato Sakai, Taro Kanao, Yohei Hamakawa, Ryo Hidaka, Masaya Yamasaki, and Kosuke Tatsumura. High-performance combinatorial optimization based on classical mechanics. *Science Advances*, 7(6):eabe7953, February 2021. ISSN 2375-2548. doi: 10.1126/sciadv.abe7953. URL https://www.science.org/doi/10.1126/sciadv.abe7953.

[15] Gaël Guennebaud, Benoît Jacob, et al. Eigen. https://libeigen.gitlab.io, 2010.

[16] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Panruo Wu, Srikara Pranesh, Stanimire Tomov, and Jack Dongarra. The design of fast and energy-efficient linear solvers: On the potential of half-precision arithmetic and iterative refinement techniques. In Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2018*, pages 586–600, Cham, 2018. Springer International Publishing. ISBN 978-3-319-93698-7.

[17] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613, 2018. doi: 10.1109/SC.2018.00050.

[18] Bruce Hajek. Cooling schedules for optimal annealing. *Mathematics of operations research*, 13(2):311–329, 1988.

[19] Koji Hukushima and Koji Nemoto. Exchange monte carlo method and application to spin glass simulations. *Journal of the Physical Society of Japan*, 65 (6):1604–1608, 1996. doi: 10.1143/JPSJ.65.1604. URL https://doi.org/10.1143/JPSJ.65.1604.

[20] IEEE Computer Society. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 2019. doi: 10.1109/IEEESTD.2019.8766229.

[21] D-Wave Systems Inc. dwave-samplers: Classical algorithms for solving binary quadratic models, 2025. URL `https://github.com/dwavesystems/dwave-samplers`.

[22] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017. doi: 10.1145/3079856.3080246.

[23] Andrew D. King, Jack Raymond, Trevor Lanting, Richard Harris, Alex Zucca, Fabio Altomare, Andrew J. Berkley, Kelly Boothby, Sara Ejtemaee, Colin Enderud, Emile Hoskinson, Shuiyuan Huang, Eric Ladizinsky, Allison J. R. MacDonald, Gaelen Marsden, Reza Molavi, Travis Oh, Gabriel Poulin-Lamarre, Mauricio Reis, Chris Rich, Yuki Sato, Nicholas Tsai, Mark Volkmann, Jed D. Whittaker, Jason Yao, Anders W. Sandvik, and Mohammad H. Amin. Quantum critical dynamics in a 5,000-qubit programmable spin glass. *Nature*, 617(7959):61–66, April 2023. ISSN 1476-4687. doi: 10.1038/s41586-023-05867-2. URL `http://dx.doi.org/10.1038/s41586-023-05867-2`.

[24] Yaohang Li, Michael Mascagni, and Andrey Gorin. A decentralized parallel implementation for parallel tempering algorithm. *Parallel Computing*, 35(5):269–283, 2009. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2008.12.009. URL https://www.sciencedirect.com/science/article/pii/S0167819108001397.

[25] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44, 2019. doi: 10.1109/HOTCHIPS.2019.8875654.

[26] Sean Lie. Cerebras architecture deep dive: First look inside the hardware/software co-design for deep learning. *IEEE Micro*, 43(3):18–30, 2023. doi: 10.1109/MM.2023.3256384.

[27] Andrew Lucas. Ising formulations of many NP problems. *Frontiers in Physics*, 2:74887, 2014.

[28] Enzo Marinari and Giorgio Parisi. Simulated Tempering: A New Monte Carlo Scheme. *Europhysics Letters (EPL)*, 19(6):451–458, July 1992. ISSN 0295-5075, 1286-4854. doi: 10.1209/0295-5075/19/6/002. URL http://arxiv.org/abs/hep-lat/9205018. arXiv:hep-lat/9205018.

[29] Benjamin Marks, Riley Collins, and Kevin C. Webb. Parallel simulated annealing with mranneal. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 545–552, 2015. doi: 10.1109/ICPADS.2015.74.

[30] Aristeidis Mastoras, Sotiris Anagnostidis, and Albert-jan N. Yzelman. Design and implementation for nonblocking execution in GraphBLAS: Tradeoffs and performance. *ACM Trans. Archit. Code Optim.*, 20(1), 3 2023. ISSN 1544-3566. doi: 10.1145/3561652. URL https://doi.org/10.1145/3561652.

[31] Tim Mattson, Timothy A. Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, Jose Moreira, and Carl Yang. Lagraph: A community effort to collect graph algorithms built on top of the graphblas. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 276–284, 2019. doi: 10.1109/IPDPSW.2019.00053.

[32] Daichi Mukunoki, Katsuhisa Ozaki, Takeshi Ogita, and Roman Iakymchuk. Conjugate gradient solvers with high accuracy and bit-wise reproducibility between cpu and gpu using ozaki scheme. In *The International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia '21, page 100–109, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450388429. doi: 10.1145/3432261.3432270. URL https://doi.org/10.1145/3432261.3432270.

[33] Lorenz K. Müller, Philippe Bich, Jiawei Zhuang, Ahmet Çelik, Luca Benfenati, and Lukas Cavigelli. Sinq: Sinkhorn-normalized quantization for calibration-free low-precision llm weights, 2026. URL https://arxiv.org/abs/2509.22944.

[34] Hiroyuki Ootomo, Katsuhisa Ozaki, and Rio Yokota. Dgemm on integer matrix multiplication unit, 2024. URL https://arxiv.org/abs/2306.11975.

[35] Muhammad Osama, Minh Truong, Carl Yang, Aydın Buluc, and John D. Owens. Graph coloring on the GPU. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 231–240. IEEE. ISBN 978-1-7281-3510-6. doi: 10.1109/IPDPSW.2019.00046. URL https://ieeexplore.ieee.org/document/8778394/.

[36] Katsuhisa Ozaki, Takeshi Ogita, Shin'ichi Oishi, and Siegfried M Rump. Generalization of error-free transformation for matrix multiplication and its application. *Nonlinear Theory and Its Applications, IEICE*, 4(1):2–11, 2013.

[37] Katsuhisa Ozaki, Yuki Uchino, and Toshiyuki Imamura. Ozaki scheme ii: A gemm-oriented emulation of floating-point matrix multiplication using an integer modular technique, 2025. URL https://arxiv.org/abs/2504.08009.

[38] J. Pawlowski, J. Tuziemski, P. Tarasiuk, A. Przybysz, R. Adamski, K. Hendzel, L. Pawela, and B. Gardas. Veloxq: A fast and efficient qubo solver, 2025. URL https://arxiv.org/abs/2501.19221.

[39] Aleksandros Sobczyk, Giuseppe Sorrentino, and Anastasios Zouzias. Segmented Operations using Matrix Multiplications, 2025. URL https://arxiv.org/abs/2506.23906. Version Number: 2.

[40] Yuki Uchino, Katsuhisa Ozaki, and Toshiyuki Imamura. Performance enhancement of the ozaki scheme on integer matrix multiplication unit. *The International Journal of High Performance Computing Applications*, 39(3):462–476, January 2025. ISSN 1741-2846. doi: 10.1177/10943420241313064. URL http://dx.doi.org/10.1177/10943420241313064.

[41] Yuki Uchino, Katsuhisa Ozaki, and Toshiyuki Imamura. Double-precision matrix multiplication emulation via ozaki-ii scheme with fp8 quantization, 2026. URL https://arxiv.org/abs/2603.10634.

[42] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. Memory performance of amd epyc rome and intel cascade lake sp server processors. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, page 165–175. ACM, April 2022. doi: 10.1145/3489525.3511689. URL http://dx.doi.org/10.1145/3489525.3511689.

[43] Bastien Vieublé. *Mixed precision iterative refinement for the solution of large sparse linear systems*. Theses, Institut National Polytechnique de Toulouse - INPT, November 2022. URL https://hal.science/tel-03975935.

[44] Deborah Volpe, Giovanni Amedeo Cirillo, Maurizio Zamboni, and Giovanna Turvani. Integration of simulated quantum annealing in parallel tempering and population annealing for heterogeneous-profile qubo exploration. *Ieee Access*, 11:30390–30441, 2023.

[45] James H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice Hall, Englewood Cliffs, NJ, 1963.

[46] Bartłomiej Wróblewski, Gioele Gottardo, and Anastasios Zouzias. Parallel Scan on Ascend AI Accelerators. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1290–1292, June 2025. doi: 10.1109/IPDPSW66978.2025.00216. URL http://arxiv.org/abs/2505.15112. arXiv:2505.15112 [cs.DC].

[47] Yinyu Ye. The gset dataset., 1996. URL https://web.stanford.edu/~yyye/yyye/Gset/.

[48] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen. A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation, 2020. URL http://albert-jan.yzelman.net/PDFs/yzelman20.pdf.

Acronyms

AI Artificial Intelligence

AIC AI Cube

AIV AI Vector

ASIC Application Specific Integrated Circuit

CG Conjugate Gradient

CPU Central Processing Unit

CSR Compressed Row Storage

dSB discrete Simulated Bifurcation

FEM Finite Element Analysis

GEMM general matrix multiply

GM Global Memory

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HPC High Performance Computing

IR Iterative Refinement

LLM Large Language Model

MTE Memory Transfer Engine

NPU Neural Processing Unit

OS Ozaki Scheme

QUBO Quadratic Unconstrained Binary Optimization

SA Simulated Annealing

SARE Simulated Annealing-Replica Exchange

SB Simulated Bifurcation

SPD Symmetric Positive Definite

SPMD Single Program Multiple Data

SpMV Sparse Matrix–Vector product

TPU Tensor Processing Unit

Acknowledgements

This thesis is the product of almost a year of work at Huawei Research. This would not have been possible if not for the guide, supervision and support of many.

First, I must thank Huawei's Computing Systems Laboratory in Zurich for welcoming me, and giving me the freedom of choosing the direction of my work. But companies are made of and by people. The recognition goes to all the colleagues working there, that have always been friendly, welcoming and helpful. The pauses playing *biliardino* were essential for the success of my work. This margin is too small to name you all!

Among the many colleagues, I am most grateful to my supervisor, Denis, who has always been there, ready to confront, support and help. You have been an invaluable guide at every step of the way.

A big thank you goes also to Albert-Jan, who, although often physically far, has always been available to help me. Your advice was fundamental for the success of my first publication!

Another person I am extremely grateful to, is my supervisor at the University of Padova: Francesco. You have always acted in the background to allow everything else to go as well as it went.

Here I must also recognize the positive influence that I have received from my family throughout these years. You have been a reliable source of support, advice but also useful distraction.

Lastly (and definitely not for importance), a thank you to all the friends with whom I shared these years. Your influence shaped the person that I am today. Too many people to personally name. I will just highlight all the friends at the *Scuola Galileiana di Studi Superiori*, with whom I shared most of my days at

university.

I will conclude with a special thank you to a special friend. Thank you, Cecilia, because you have been a source of support, especially when I needed it most.