Saint Petersburg Electrotechnical University
(ETU)

Education profile	09.04.01 Computer Science and Engineering
Education program	Computer Science and Knowledge Discovery
Faculty	Computer Science and Technology
Department	Information Systems

Accepted for defense

Head of the department V. V. Tcehanovsky

MASTER'S GRADUATE QUALIFICATION WORK

Topic: ALGORITHMS FOR OPTIMIZING SIMD-INSTRUCTIONS IN MODERN ARCHITECTURES

Student		*signature*	Haitao Mei
Supervisor	(PhD. Associate Professor)	*signature*	A. A. Paznikov
Advisors	(D.E. Prof.)	*signature*	I. V. Medynskaya
		signature	O. I. Brikova
	(PhD. Associate Professor)	*signature*	N. A. Nazarenko

Saint Petersburg

2023

TASK FOR THE GRADUATE QUALIFICATION WORK

Approved

Head of the department Systems department

____________ V. V. Tcehanovsky

«___»______________20___.

Student Haitao Mei Group 7300

Topic: Algorithms for optimizing SIMD-instructions in modern architectures

Institution: Saint Petersburg Electrotechnical University (LETI)

Initial data (technical requirements):

Algorithms for optimizing SIMD-instructions in modern architectures

Contents:

Bibliography review

Foundations of modern compiler

Theory of loop tiling optimization

Algorithm programming Conclusion and future plan

Commercialization of the Research's Result

List of reporting materials: the text of the GQW, illustrations, other reporting materials

Additional sections: Commercialization of the Research's Results

The task was given	The GQW was submitted for the defense
«___»______________20___.	«___»______________20___.

Student	__________________	Haitao Mei
Scientific advisor *(Academic degree, title)*	___________________	A. A. Paznikov

CALENDER PLAN FOR THE GRADUATE QUALIFICATION WORK

Approved

Head of the department FKTI

_____________V. V. Tcehanovsky

«___»______________2023 .

Student Haitao Mei Group:7300
Topic: Algorithms for optimizing SIMD-instructions in modern architectures

Nº	Stages	Deadline
1	Bibliography review	31.03 – 07.04
2	Foundations of modern compiler	08.04 – 16.04
3	Theory of loop tiling optimization	17.04 – 25.04
4	Algorithm programming	26.04 – 10.05
5	Conclusion and future plan	10.05 – 12.05
6	Commercialization of the Research's Results	10.05 – 12.05

Student	____________________	Mei Haitao
Scientific advisor *(Phd)*	____________________	A. A. Paznikov

SUMMARY

Explanatory note 82 p., 27 fig., 13 tab., 11 refernce..

KEYWORDS: LOOP TILING, SIMD, COMPILIER, PARALLEL, CACHE OPTIMIZATION

The subject of the research is: Algorithms for optimizing SIMD-instructions in modern architectures

The target of the GQW – Conduct research and analysis on modern compiler's automatic vectorization algorithms using the TSVC test suite, identify the weaknesses of automatic vectorization, understand the underlying reasons, and implement optimizations.

Modern CPUs support vectorized computation. Loop tiling can improve cache data reuse and reduce waiting time, thus minimizing cache misses and allowing SIMD to work efficiently. Existing loop tiling techniques have achieved good performance, but they mainly consider the performance acceleration when the loop scale is a multiple of the vectorization width, neglecting the case when the loop scale is not a multiple of the vectorization width. Such situations may be encountered in fields like AI, where loop blocking can lead to misaligned cache addresses.

This thesis proposes an algorithm that estimates vectorization benefits in advance, calculating the maximum gains in non-vectorized width situations. The aim is to improve program performance in these specific cases.

ABSTRACT

In this thesis, we present a comprehensive summary of a two-year research journey on the optimization of Single Instruction Multiple Data (SIMD) instructions in modern computer architectures. The first year focused on compiler-level SIMD optimization, while the second year extended the scope to loop optimization within polyhedral models. Our research delves into the history, hotspots, and bottlenecks of this field, providing insights into the development of parallel computing and its inherent challenges.

Nested loops are prevalent in computationally intensive domains, where traditional SIMD optimization techniques often target loop iteration counts that are multiples of 4. However, in the AI domain, non-multiple of non-vectorized width, such as N=7, are commonly encountered. To address this issue, we introduce the polyhedral model and apply loop tiling techniques for optimization. Loop optimization serves to enhance data locality, reduce cache misses, and increase code parallelism, fully exploiting the multi-core capabilities of modern processors.

In this thesis, we replicate and optimize an existing algorithm that estimates the performance gains achievable through vectorization and calculates the optimal tile size accordingly. Our improved approach demonstrates significant advantages in enhancing the performance of computationally intensive tasks, achieving noticeable performance improvements across various test cases compared to traditional methods.

TABLE OF CONTENTS

DEFINITIONS, DESIGNATIONS AND ABBRIVIATIONS

The present explanatory note uses the following abbreviations and designations:

LLVM– Low Level Virtual Machine

Pluto- An automatic parallelizer and locality optimizer foe affine loop nests

OpenMP - Open Multi-Processing

SIMD - Single Instruction Multiple Data

INTRODUCTION

The rapid development of computer technology has led to an increasing demand for high-performance computing, particularly in areas such as artificial intelligence, scientific simulations, and multimedia processing. Single Instruction Multiple Data (SIMD) instructions in modern computer architectures have emerged as a vital technique for enhancing the performance of computationally intensive tasks by exploiting data-level parallelism. However, effectively utilizing SIMD instructions and optimizing their performance remains a challenging task, especially when dealing with complex loop structures and non-multiple of 4 iteration counts, which are commonly encountered in the AI domain.

To address these challenges, we turn our attention to the polyhedral model, a powerful framework for representing and optimizing multidimensional loop structures. The polyhedral model captures loop nests and their dependencies as polyhedra, enabling advanced analysis and transformation of loops for performance optimization. Loop tiling is a well-established polyhedral transformation technique that decomposes loops into smaller sub-blocks, improving both data locality and parallelism. By combining the polyhedral model with SIMD parallelization strategies, we aim to achieve higher performance gains.

This thesis summarizes our two-year research journey, which initially focused on compiler-level SIMD optimization in the first year and later extended to loop optimization within polyhedral models, including parameterized tiled loops in OpenGL, in the second year. We replicate and optimize an existing algorithm that estimates the performance improvements achievable through vectorization and calculates the optimal tile size accordingly. Our improved approach demonstrates significant advantages in enhancing the performance of computationally intensive tasks, achieving noticeable performance improvements across various test cases compared to traditional methods.

The remainder of this thesis is organized as follows: Section 2 reviews the relevant literature on SIMD parallelization, polyhedral models, loop tiling, and parameterized tiled loops in OpenGL. Section 3 presents the fundamentals of the loop tiling algorithm and describes our optimized approach. Section 4 discusses hardware-specific optimizations and strategies to further enhance SIMD parallelization. Section 5 provides an experimental evaluation of our approach, including comparisons with traditional methods. Finally, Section 6 concludes the thesis and outlines future research directions.

1. THE MODERN STATE OF THE PROBLEM UNDER STUDY

Loop tiling is a widely-used loop optimization technique that applies affine transformations to reorganize nested loop segments within a program. This method serves a dual purpose: enhancing data locality to reduce cache miss rates and uncovering coarse-grained parallelism within the loop code, which enables the efficient use of multi-core processors. After the tiling process, the loop iteration order is adjusted based on tile size, effectively minimizing the data reuse distance. Consequently, choosing an appropriate tile size is crucial for the performance of tiled loop code.

Traditional tile size selection (TSS) algorithms have primarily focused on improving program locality and exposing coarse-grained parallelism. In contrast, more recent TSS algorithms also consider fine-grained parallelism as an additional optimization target. For example, Feld et al. [1] addressed unaligned data access by ensuring that the tile size of vectorizable loop levels is a multiple of the vector factor (i.e., the maximum number of operands that can be stored in a vector register). They also aimed to optimize the outer loop's tile size by analyzing memory access patterns, thus maximizing L1 cache utilization.

Mehta et al. [2] studied their TSS model and found that loop tiling and vectorization could have conflicting effects in certain nested loops. While loop tiling divides loop iterations, vectorization requires longer loop iterations for optimal performance. As a result, they chose not to apply tiling to vectorizable loop levels. Subsequently, Mehta et al. [3] introduced an improved TTS model for nested loops characterized by large memory access arrays, enabling the full exploitation of microprocessors' data prefetching capabilities.

However, these algorithms do not provide a quantitative analysis of the impact that varying tile sizes have on vectorization benefits. Additionally, they primarily focus on nested loops where memory access array sizes are integer multiples of the vector factor. For nested loops that do not meet this criterion， these studies do not offer a comprehensive analysis.

2. FUNDAMENTALS OF MODERN COMPILER

2.1. Introduction to modern compilers

Traditional compilers are usually divided into three parts: front-end, optimizer, and back-end. In the compilation process, the front-end is mainly responsible for lexical and syntactic analysis, converting source code into abstract syntax trees. The optimizer, based on the front-end, optimizes the obtained intermediate code to make it more efficient. The back-end then translates the optimized intermediate code into machine code specific to each platform.

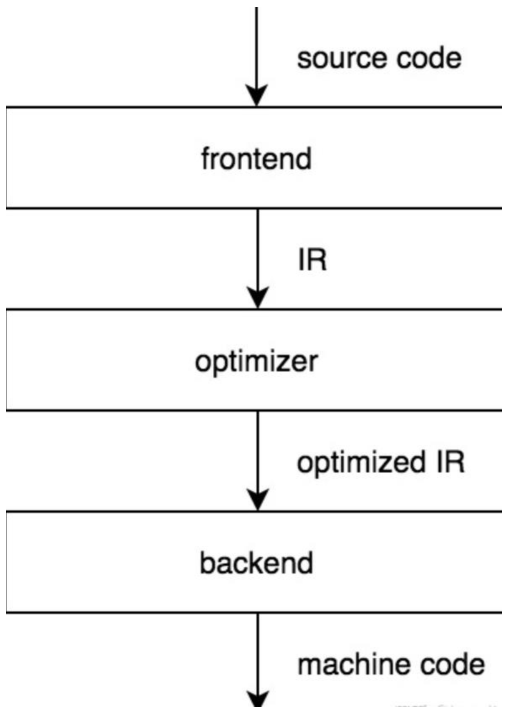


Figure 1 - compiler structure

2.1.1. GCC (GNU Compiler Collection)

GCC is an open source compiler suite developed by the GNU Project that supports a variety of programming languages including C, C++, Objective-C, Fortran, Ada, D, and others. It is widely used for its portability, versatility and wide range of optimizations.

Figure 2 – GCC’s logo

The main features and optimizations of GCC include.

A. Auto-vectorization: GCC automatically identifies and vectorizes loops that could benefit from SIMD parallelism, thus improving the performance of modern processors.

B. Graphite: This is an advanced loop transformation framework based on the polyhedral model that enables complex loop optimizations to enhance data locality and parallelism.

C. Link Time Optimization (LTO): LTO allows GCC to perform inter-process optimizations such as inlining, constant propagation, and dead code elimination during the link phase to produce more efficient code.

D. Profile-guided optimization (Pgo): Pgo uses analysis information collected during program execution to make better optimization decisions, customizing the generated code to the specific usage patterns of the program.

2.1.2. LLVM (low-level virtual machine)

LLVM is a widely used open source compiler infrastructure that provides a modular and reusable toolkit for building compilers, optimizers, and runtime environments. It supports a variety of front- and back-ends for different programming languages and architectures. Compiler-related support is provided, enabling compile-time optimization of programming languages, link optimization, inline compilation optimization, and code generation. In short, it can be used as a backend for a variety of compilers.

Figure 3 – llvm's logo

In 2000, Chris Lattner, a recent college graduate, took the regular path of the GRE and eventually went to the University of Illinois at Urbana-Champaign to begin a difficult journey in computer science, pursuing his master's and PhD degrees. During this period, he not only traveled to various sites in the US, but also became an expert in the field of compilers by studying "Compilers: Principles, Techniques, and Tools" in depth. He continued to explore the uncharted territory of compilers and published one paper after another. In his master's thesis, he laid the foundation for LLVM by proposing a complete set of compiler ideas for optimizing programs at compile time, link time, run time, and even idle time. LLVM matured during his PhD research, using GCC as a front-end to semantically analyze user programs and generate intermediate formats (IF). L

LVM then used the analysis results to complete code optimization and generation. This research led him to become an industry-renowned compiler expert by the time he graduated in 2005, and he was soon targeted by Apple, eventually becoming a stalwart of its compiler project.

Key features and optimizations of LLVM include.

A. Clang: This is a high performance c/C++/Objective-C front-end for LLVM, known for its fast compile times, excellent diagnostics and compatibility with GCC.

B. LLVM Intermediate Representation (LLVM IR): LLVM uses a low-level, typed and platform-independent intermediate representation that enables powerful optimizations and simplifies the development of new compiler code.

C. Loop Optimization: LLVM includes a comprehensive set of loop optimizations such as loop unrolling, loop invariant code movement, and loop fusion that improve performance and enable further SIMD and cache optimizations.

D. Machine-level optimizations: The back end of LLVM performs aggressive instruction scheduling, register allocation, and peephole optimizations to generate efficient machine code for the target architecture.

2.1.3. ICC (Intel C++ Compiler)

ICC is a commercial compiler developed by Intel specifically to optimize performance on Intel processors. It supports the C, C++, and Fortran programming languages and provides numerous optimizations to take full advantage of Intel's architectural capabilities.

Figure 4 - intel logo

Intel developed the ICC compiler to meet the specific needs of its processor architectures and customers, focusing on delivering superior performance and a seamless development experience. While GCC and LLVM are powerful general-purpose compilers, ICC offers clear advantages in areas such as automatic vectorization, parallel programming support, and optimizations for Intel-specific features. As the technology landscape continues to evolve, it is expected that Intel will continue to enhance ICC to stay ahead of the curve and provide its customers with the best performance and development experience.

Key features and optimizations of ICC include.

A. Advanced Vector Extensions (AVX) support: ICC generates code that leverages Intel's AVX, AVX2, and AVX-512SIMD instruction sets, significantly improving the performance of data parallel operations.

B. Intraprocedural Optimization (IPO): ICC performs aggressive IPOs such as function inlining, cross-file constant propagation, and dead code elimination to optimize code across function and file boundaries.

C. Profile-guided optimization (PGO): Similar to GCC, ICC uses performance analysis information to make more informed optimization decisions, customizing the generated code to real-world usage patterns.

D. Threading and parallelism: ICC supports OpenMP, Intel Threading Building

Block (TBB) and cilk Plus for efficient parallel programming, providing automatic parallelization and vectorization to take advantage of multi-core and SIMD capabilities.

2.2. Compiler Optimization Methods

Compilers play a crucial role in the execution and optimization of programs, and modern compilers combine various complex techniques to improve performance. This paragraph provides a brief overview of the most prominent optimization strategies in modern compilers, including cache optimization, SIMD (Single Instruction, Multiple Data) optimization, and polyhedral optimization.

2.2.1. Cache Optimization

Cache optimization aims to minimize cache misses and improve data locality by reorganizing memory accesses in a program. By optimizing cache usage, compilers can significantly reduce execution time and improve overall performance. Key cache optimization techniques and strategies employed by GCC, Clang, and ICC include:

a. Loop Interchange: Reorders nested loops to improve spatial and temporal locality, ultimately reducing cache misses. GCC and Clang both utilize loop interchange to optimize cache usage. ICC also employs this technique, often providing more aggressive loop interchange optimizations.

b. Loop Tiling: Divides loop iterations into smaller chunks (tiles) to take advantage of cache hierarchy and improve data reuse. GCC, Clang, and ICC all implement loop tiling optimizations, with each compiler offering different heuristics for determining tile sizes and shapes.

c. Loop Fusion: Combines adjacent loops with similar iteration space, reducing cache misses by reusing data loaded into the cache. GCC, Clang, and ICC all support loop fusion optimization, using various heuristics to determine when loop fusion is beneficial.

2.2.2. SIMD Optimization

SIMD optimization leverages the parallel processing capabilities of modern processors, which can execute a single instruction on multiple data elements simultaneously. By vectorizing code and using SIMD instructions, compilers can significantly improve the performance of data-parallel operations. Key SIMD optimization techniques and strategies employed by GCC, Clang, and ICC include:

a. Auto-Vectorization: Automatically identifies parallelizable loops and converts them into SIMD instructions, increasing the throughput of operations without requiring manual code modifications. GCC, Clang, and ICC all support auto-vectorization, with each compiler implementing unique heuristics for identifying vectorizable loops.

b. Manual Vectorization: Allows developers to explicitly specify SIMD operations using intrinsic functions or directives, providing fine-grained control over vectorization. GCC, Clang, and ICC all support manual vectorization, offering varying levels of control and optimization for SIMD operations.

c. Runtime Code Generation: Adapts code to the specific SIMD capabilities of the target hardware, enabling performance portability across different architectures. ICC is particularly known for its robust runtime code generation capabilities, while GCC and Clang also provide some support for adapting to target hardware.

2.2.3. Polyhedral Optimization

Polyhedral optimization is a powerful technique for optimizing loop nests with affine control structures and data accesses. By representing loop nests and their dependencies as geometric objects in a high-dimensional space, compilers can apply a range of advanced transformations to improve parallelism, data locality, and overall performance. Key polyhedral optimization techniques and strategies employed by GCC, Clang, and ICC include:

a. Loop Transformations: Applies affine transformations to loop nests, such as loop interchange, loop skewing, and loop reversal, to enhance parallelism and data locality. GCC includes the Graphite framework for polyhedral optimizations, while ICC has its own polyhedral optimization infrastructure. Clang currently lacks native support for polyhedral optimizations but can utilize the Polly plugin to enable them.

b. Loop Tiling: Applies loop tiling within the polyhedral model, dividing loop iterations into smaller chunks (tiles) that can be processed more efficiently in cache. Both GCC's Graphite framework and ICC's polyhedral optimization infrastructure support loop tiling in the polyhedral model.

d. Pipelining: Exploits coarse-grained and fine-grained parallelism by overlapping the execution of different loop iterations, increasing throughput and reducing latency. ICC is known for its advanced loop pipelining capabilities, while GCC's Graphite framework also supports pipelining in the polyhedral model. Clang, when used with the Polly plugin, can achieve similar optimizations.

2.3. Comparison of the three compilers

Modern compilers employ a range of advanced optimization strategies to enhance program performance. Cache optimization, SIMD optimization, and polyhedral optimization are among the most important techniques in this field, and their combined use can significantly improve execution time, parallelism, and data locality. As hardware and software technologies continue to evolve, compilers will undoubtedly adopt.

Apple has always used GCC as the official compiler. As an open source compiler, GCC has always performed well, but Apple requires more from its compilation tools. There are two main reasons for this.

First, Apple added many features to the Objective-C language (including later additions to C), but GCC developers were not interested in implementing them. As a result, the two parties eventually split and developed on separate branches, resulting in Apple's version of the compiler lagging far behind the official GCC version.

Second, GCC's code was too coupled and difficult to isolate, and the quality of higher versions of the code deteriorated. However, many of the features Apple wanted to implement (such as better IDE support) required a modular approach to calling GCC, which GCC had always refused to do.

Immediately after joining Apple, Chris Lattner demonstrated his expertise by first working on code optimizations within the OpenGL team to implement LLVM runtime compilation on the OpenGL stack, enabling the OpenGL stack to generate more efficient graphics code. If the graphics card is advanced enough, this code will be executed directly on the GPU. However, for some graphics cards that did not support all OpenGL features (such as the Intel GMA cards of the time), LLVM could optimize these instructions into efficient CPU instructions that would allow the program to run properly. This powerful OpenGL implementation was used in the later release of Mac OS X10.5. Also, LLVM's linking optimizations were added directly to Apple's code linker, and LLVM-GCC uses GCC 4.0 code in parallel.

LLVM's strengths lie precisely in addressing the weaknesses of GCC. Traditional compilers work in three stages: the frontend is responsible for parsing source code, checking for syntax errors, and translating it into an abstract syntax tree (AST). The optimizer then improves the intermediate code, striving for greater efficiency. The backend is tasked with converting the optimized intermediate code into target machine code, maximizing the use of the target machine's specialized instructions to enhance code performance. In fact, this model applies not only to static languages but also to dynamic languages like Java. The JVM employs the same model, translating Java code into Java bytecode.

One advantage of this model is that when multiple languages need to be supported, adding multiple frontends will suffice. Likewise, when supporting various target machines, only multiple backends need to be added. For the intermediate optimizer, a universal intermediate code can be utilized. This three-stage structure has another benefit: frontend developers only need to know how to convert source code into intermediate code that the optimizer can understand, without needing knowledge of the optimizer's inner workings or the target machine. This significantly reduces the difficulty of compiler development, allowing more developers to participate.

Despite the many advantages of this three-stage compiler, which has been included in textbooks, it has never been perfectly implemented in practice. Java and .NET virtual machines are among the best examples. Virtual machines can translate target languages into bytecode, so theoretically, any language can be translated into bytecode and run within the virtual machine. However, this dynamic language model is not well-suited for the C language, and forcibly translating C language into bytecode while implementing a garbage collection mechanism is highly inefficient. GCC has done a relatively good job with the three-stage model and has implemented numerous frontends to support various languages. nonetheless, the critical drawback of the aforementioned compilers is that they exist as a single, complete executable, without providing an interface for code reuse by developers of other languages. Even though GCC is open-source, reusing its source code can be quite challenging.

As a leading company in microprocessors and technology, Intel recognized the need to develop its own compiler, the Intel C++ Compiler (ICC), to take full advantage of its processors' capabilities and meet its customers' performance requirements. Several reasons led to the development of ICC, and the following paragraphs delve into its story and compare its advantages over GCC and LLVM.

One of the main motivations for developing ICC was to optimize the code specifically for Intel processors. While both GCC and LLVM are general purpose compilers, they may not take full advantage of the unique features and capabilities of Intel processors. By creating the ICC, Intel can ensure that the generated code takes advantage of the latest architectural advances and instruction sets in its processors, such as the AVX-512 extensions.

In addition to architectural benefits, Intel seeks to provide its customers with a seamless development experience by offering the ICC in the Intel Parallel Studio xe suite, as well as other software tools. This Integrated Development Environment (IDE) includes performance libraries, analysis tools and debuggers, all designed to work in concert with ICC to further improve application performance on Intel platforms.

In addition, ICC is known for its advanced auto-vectorization feature that automatically identifies and converts sections of code that can benefit from SIMD instructions. While GCC and LLVM also offer auto-vectorization, ICC's performance in this area is often considered superior, especially for Intel processors.

Another key advantage of ICC over GCC and LLVM is its extensive support for parallel programming. ICC provides advanced support for OpenMP, a widely used parallel programming model that allows developers to write multi-threaded applications more easily. While OpenMP is also supported by GCC and LLVM, ICC implementations are typically more mature and better optimized for Intel hardware.

3. THEORY OF LOOP TILING OPTIMIZATION

3.1. Basic optimization knowledge

3.1.1. Cache theory

Software developers often wish for computer hardware to have infinite capacity, zero access latency, unlimited bandwidth, and inexpensive memory. However, the reality is that the larger the memory capacity, the longer the corresponding access time; the faster the memory access speed, the more expensive it becomes; and the greater the bandwidth, the higher the price. To resolve the contradiction between large capacity, high speed, and low cost, the principle of data locality is applied, placing frequently accessed data in small, high-speed storage devices and organizing storage devices with different speeds into hierarchical layers that work together.

Modern computer storage hierarchies can be divided into several layers. As shown in the diagram below, registers are located within the processor; slightly further away is the Level 1 (L1) Cache, which can generally store 64 KB of data and takes about 1 ns to access. The L1 Cache is typically divided into an instruction cache (from which the processor retrieves instructions to execute) and a data cache (from which the processor stores/retrieves operands for the instructions). Next is the Level 2 (L2) Cache, which usually stores both instructions and data, has a capacity of approximately 256 KB, and takes about 3-10 ns to access. Following that is the Level 3 (L3) Cache, with a capacity of around 16-64 MB and an access time of about 10-20 ns. Further down the hierarchy are main memory, hard drives, and so on. Note that the CPU and Cache transfer data in words, while the Cache to main memory transfers data in blocks (typically 64 bytes).

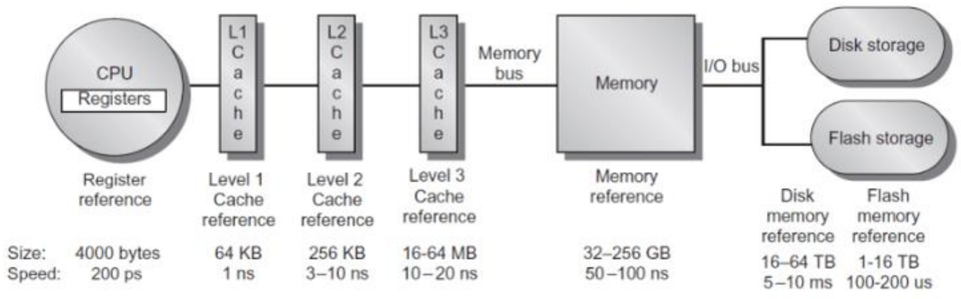


Figure 5 - cache structure

The principle of data locality mentioned earlier generally refers to temporal locality (when a program accesses the same memory space multiple times within a certain period) and spatial locality (when a program accesses nearby memory spaces within a certain period). The efficiency of high-speed cache (Cache) depends on the spatial and temporal locality properties of the program. For example, when a program repeatedly executes a loop, ideally, the first iteration will bring the code into the cache, and subsequent iterations will access data directly from the cache without needing to reload from main memory. Therefore, to achieve better performance, data accesses should occur in the cache as much as possible. However, if conflicts occur when accessing data in the cache, performance may decrease.

3.1.1.1. Alignment and Layout

Modern compilers can improve the cache hit rate and program performance by adjusting the layout of code and data. This section mainly discusses the impact of data and instruction alignment and code layout on program performance. In most processors, Cache-to-main memory transfers use Cache lines (generally 64 bytes, sometimes called Cache blocks; this article will use Cache lines consistently). The CPU loads data from memory one Cache line at a time and writes data to memory one Cache line at a time.

Suppose the processor accesses a data object A for the first time, and its size is exactly 64 bytes. If the starting address of data object A is not aligned, i.e., the data object A occupies parts of two different Cache lines, the processor needs two memory accesses when accessing this data object, resulting in lower efficiency. However, if data object A is memory-aligned, i.e., it is entirely within one Cache line, the processor only needs one memory access when accessing the data, resulting in much higher efficiency. The compiler can arrange data objects reasonably to avoid unnecessarily spanning them across multiple Cache lines and concentrate the same object within a single Cache, thereby effectively using the Cache to improve program performance. This can be achieved by sequentially allocating objects; if the next object cannot fit into the remaining portion of the current Cache line, skip the remaining portion and allocate the object from the beginning of the next Cache line, or by allocating objects of the same size in the same storage area, with all objects aligned on size-multiple boundaries.

Cache line alignment may lead to wasted storage resources, as shown in Figure 2, but execution speed may improve as a result. Alignment can be applied not only to global static data but also to data allocated on the heap. For global data, the compiler can use assembly language alignment instructions to notify the linker. For heap-allocated data, placing objects on Cache line boundaries or minimizing the number of times objects cross Cache lines is not the responsibility of the compiler but rather the storage allocator in the runtime.

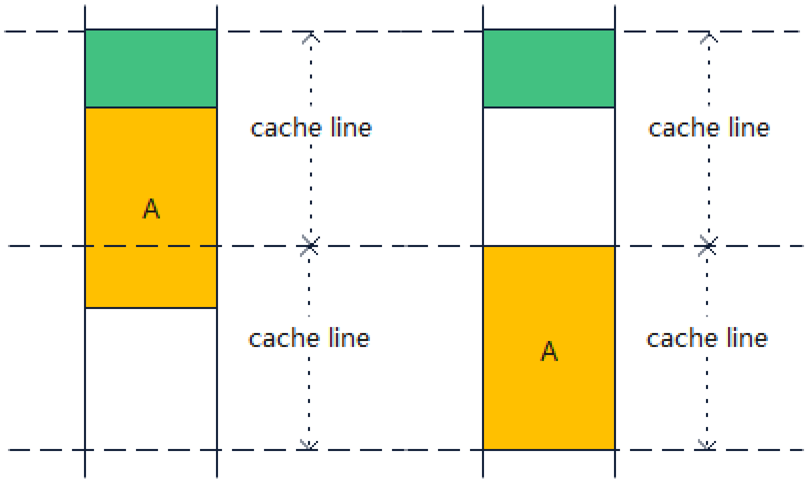


Figure 6 - cache line

3.1.1.2. Utilizing Hardware Assistance

Cache prefetching involves loading instructions and data from memory into the Cache in advance, with the aim of speeding up processor execution. Cache prefetching can be implemented through hardware or software. Hardware prefetching is realized through dedicated hardware units within the processor, which predict the memory addresses to be accessed by tracking the change patterns of memory access instruction data addresses and read the data from main memory into the Cache beforehand. Software prefetching involves explicitly inserting prefetch instructions in the program, allowing the processor to read data from memory to Cache in a non-blocking manner. Since hardware prefetchers typically cannot be dynamically turned off, in most cases, software prefetching and hardware prefetching coexist, and software prefetching must strive to cooperate with hardware prefetching to achieve better results.

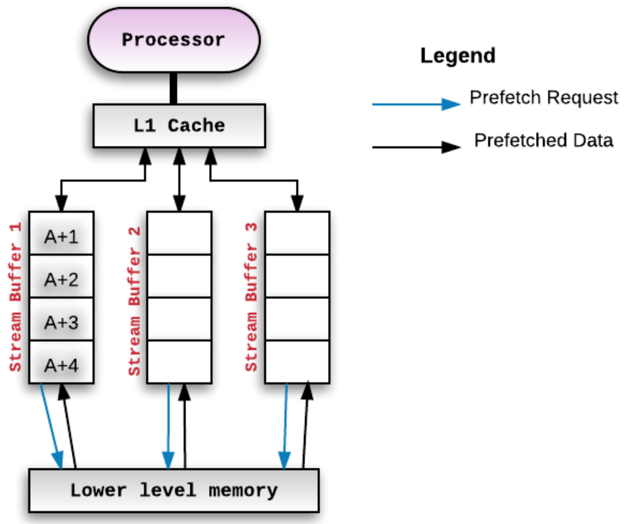


Figure 7 - cache prefetching structure

The prefetch instruction prefetch(x) is a hint that informs the hardware to start loading data from address x in main memory into the Cache. It does not cause the processor to stall, but if the hardware detects that an exception would occur, it will ignore this prefetch operation. If prefetch(x) is successful, it means that the next time x is accessed, it will hit the Cache. An unsuccessful prefetch operation may result in a Cache miss during the next read but will not affect the correctness of the program.

It is important to note that prefetching does not reduce the latency of fetching data from main memory to Cache. Instead, it hides this latency by overlapping prefetching with computation. In summary, prefetching is applicable when the processor has prefetch instructions or non-blocking read instructions that can be used for prefetching, when the processor cannot dynamically reorder instructions or when the dynamic reordering buffer is smaller than the specific Cache latency we want to hide, and when the data in question is larger than the Cache or cannot be determined if it is already in the Cache. Prefetching is not a panacea, and improper prefetching may cause Cache conflicts and reduce program performance. We should first utilize data reuse to reduce latency and then consider prefetching.

3.1.1.3. Loop tiling

Loop tiling is a method to optimize loop structures, including swapping two nested loops (loop interchange), reversing the loop iteration execution order (loop reversal), merging two loop bodies into one (loop fusion), loop splitting (loop distribution), loop tiling (loop blocking), and loop unrolling and jamming, among others. The main purpose of these transformations is to optimize the use of registers, data caches, and other storage levels.

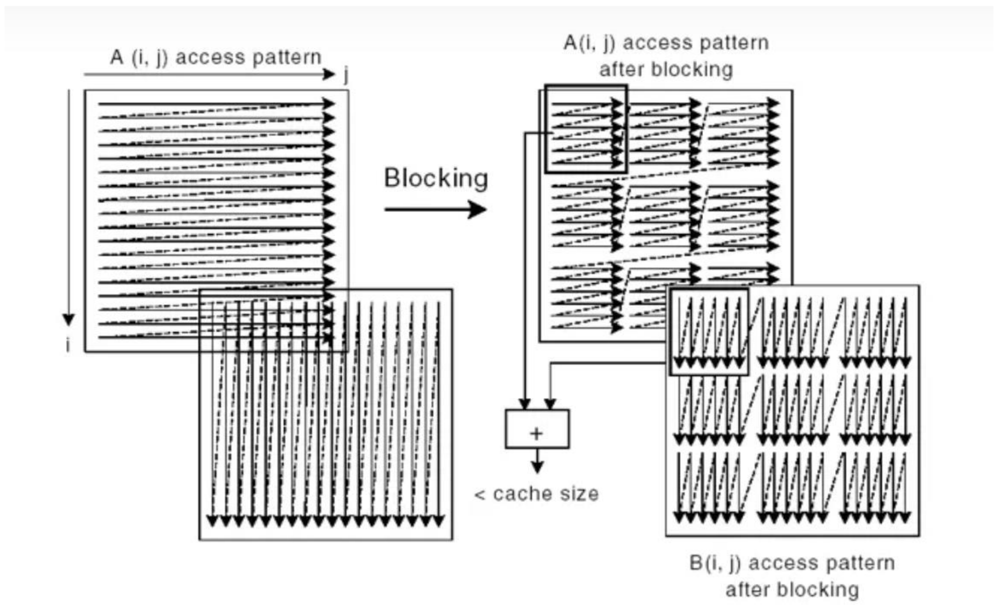

A (i, j) access pattern
j
i
Blocking
A(i, j) access pattern after blocking
+
< cache size
B(i, j) access pattern after blocking

Figure 8 - Common loop tiling algorithms

This article focuses primarily on loop tiling, a method that divides the loop's iteration space into smaller sub-blocks, allowing computation to take place on smaller data sets. Loop tiling offers the following advantages:

1. Improve locality: Loop tiling improves data access locality by dividing data into smaller subsets. This means that when processing subsets, the processor is more likely to hit data in the cache, reducing memory access latency.
2. Increase Cache hit rate: Loop tiling divides the computation process into smaller sub-blocks, making the data in these sub-blocks easier to load into the cache and reuse during loop execution. This significantly improves the Cache hit rate, reduces Cache misses, and enhances program performance.
3. Reduce Cache replacement: Loop tiling helps to limit calculations to smaller data sets, reducing data replacement in the cache. This means that during sub-block execution, the probability of data being replaced

in the Cache is lower, further reducing memory access delay.

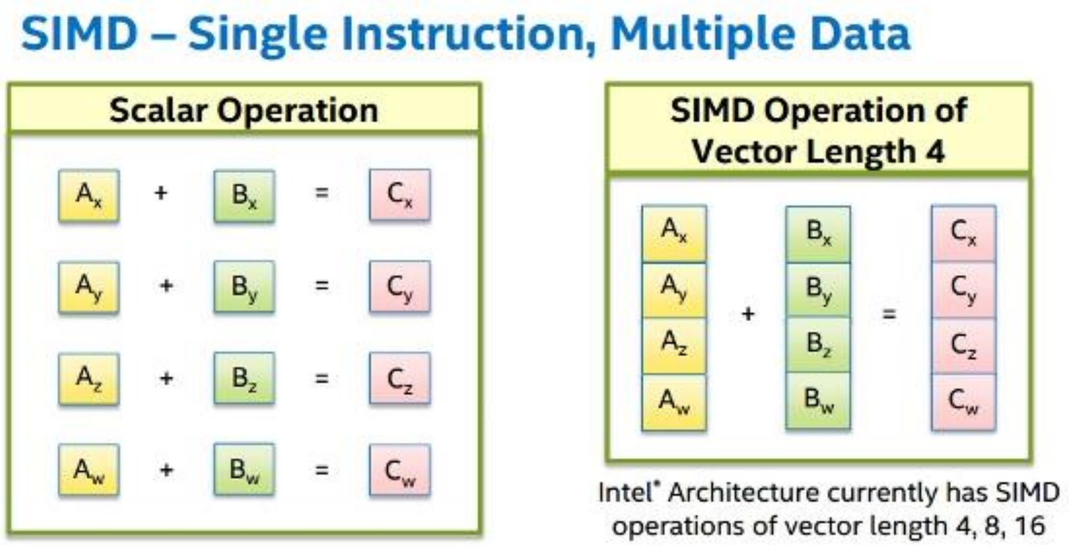
4. Optimize multi-level Cache utilization: Modern computer systems typically feature a multi-level Cache structure. Loop tiling technology can adjust the size of sub-blocks based on different cache capacities and access latencies, achieving better data reuse between multi-level caches.
5. Improve parallelism: After loop tiling, computation can be performed on smaller sub-blocks. This provides the possibility to utilize computing resources such as multi-core processors, vector instructions, or GPUs. Based on loop tiling, program execution performance can be further improved by optimizing parallelism strategies.

For any loop compilation, we need to discuss its legality (preserving the semantics of the program, or preserving data dependency for loops in this context) and profitability (how much cache miss is reduced after transformation, etc.). For the strip-mine-and-interchange blocking method we introduced above, the strip-mine part (block execution) is always legal, but the interchange part (moving the inner loop to the outer layer) is not necessarily legal and requires some analysis. For the profitability part, we have already introduced how to calculate cache miss in the previous section.

3.1.2. SIMD theory

SIMD (Single Instruction, Multiple Data) is a type of parallel computing. It allows hardware to perform calculations on multiple data elements simultaneously using a single instruction.

As shown in the illustration, the advantage of SIMD is that it can optimize code execution by reducing the number of required instructions. Instead of using four instructions to obtain the results, SIMD optimization allows for the same results to be achieved with just one instruction.


SIMD – Single Instruction, Multiple Data
Scalar Operation
A_x + B_x = C_x
A_y + B_y = C_y
A_z + B_z = C_z
A_w + B_w = C_w
SIMD Operation of Vector Length 4
Intel® Architecture currently has SIMD operations of vector length 4, 8, 16
Figure 9 - SMID principle

SIMD was invented in the 1970s, but it wasn't until the 1990s when graphics cards were developed that SIMD truly began to shine. Each processor manufacturer has its own set of SIMD instruction sets:

- Intel (MMX/SSE/AVX)
- AMD (3DNow!)
- MIPS (MDMX)
- Arm (NEON)

Compilers provide us with C language built-in functions to call SIMD instructions, so we don't need to study the instructions of each platform. Although the internal implementation of each compiler may have many differences, these built-in functions make it easier for developers to leverage SIMD capabilities across different platforms.

3.1.3. Polyhedral theory

Polyhedral optimization is a compiler optimization technique used to improve the performance of nested loop codes. It is based on the polyhedral model, which represents programs by mapping loop iteration spaces to high-dimensional integer point sets. This representation allows the compiler to perform global optimizations in complex loop structures, such as loop transformations, improving data locality, and exploiting parallelism.

Polyhedral optimization has the following main features:

- Abstract representation: Polyhedral optimization abstracts the loop structures and data dependencies in the code into geometric structures, making complex code analysis and transformations simpler and more intuitive.

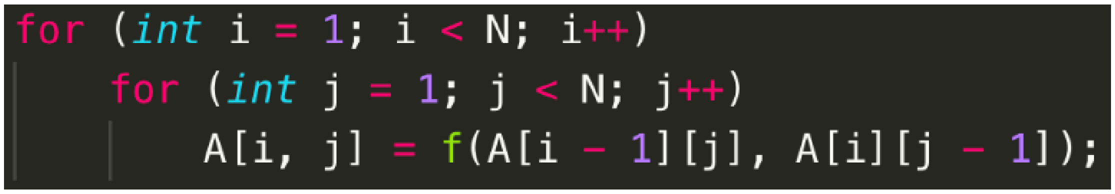
- Loop transformations: Through the polyhedral model, the compiler can perform various loop transformations, such as loop interchange, loop skewing, loop tiling, etc., to improve program performance.

- Data locality: Polyhedral optimization focuses on improving the locality of data accesses

3.1.3.1. Principle introduction

To provide a simple introduction to the working principles of Poly for readers who may not be familiar with the concept, we'll try to explain the basic principles of Poly using straightforward algebraic and geometric descriptions. This explanation is based on the illustrations found in reference [5], which we will interpret for better understanding.

Let's start by considering a simple nested loop, as shown in Figure 1, where N is a constant. The statement inside the loop updates the data at the $A[i][j]$ position by referencing stored data at $A[i-1][j]$ and $A[i][j-1]$. If we represent each iteration instance of the statement within the loop as a point in space, we can construct a two-dimensional space based on (i, j), as illustrated in Figure 11. Each black point in the diagram represents an iteration instance of the statement that writes to $A[i][j]$. Consequently, we can create a rectangle consisting of all the black points, which can be regarded as a Polyhedron in two-dimensional space. This space is referred to as the iteration space for this computation.



```
for (int i = 1; i < N; i++)
    for (int j = 1; j < N; j++)
        A[i, j] = f(A[i - 1][j], A[i][j - 1]);
```

Figure 10 - loop Analysis

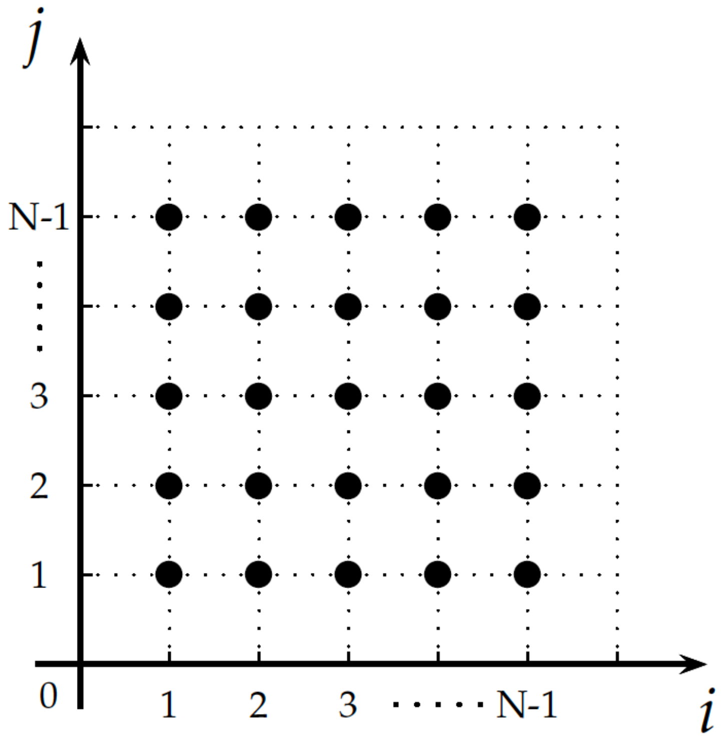


Figure 11 – loop space

We can represent the Polyhedron in this two-dimensional space using the set notation in algebra: $\{[i, j] : 1 <= i <= N - 1 \, and \, 1 <= j <= N - 1\}$, where $[i, j]$ is a pair, and the inequalities following the colon indicate the range of the set. We can assign a name to this pair, such as S, which represents a statement. The Polyhedron of this statement can then be expressed as $\{S[i, j] : 1 <= i <= N - 1 \, and \, 1 <= j <= N - 1\}$.

Since statement S iterates through the i loop first and then the j loop, we can define a schedule (order) for statement S. This schedule is represented by the mapping $\{S[i, j] -> [i, j]\}$, which indicates that statement $S[i, j]$ iterates in the order of i, followed by the order of j.

Next, let's analyze the relationship between the statement and the accessed array.

In algebra, we use mappings to represent this relationship. In Figure 10, statement S reads and writes to array A, so we can use Poly to calculate the read-access relationship between S and A, which can be expressed as $\{S[i,j] -> A[i-1,j] : 1 <= i <= N - 1 \; and \; 1 <= j <= N - 1; S[i,j] -> A[i,j-1] : 1 <= i <= N - 1 \; and \; 1 <= j <= N - 1\}$. Similarly, the write-access relationship can be expressed as $\{S[i,j] -> A[i,j] : 1 <= i <= N - 1 \; and \; 1 <= j <= N - 1\}$.

Based on these read and write access relationships, Poly can calculate the dependency relationship within the nested loop. This dependency relationship can be expressed as another mapping, which is $\{S[i,j] -> S[i, 1 + j] : 1 <= i <= N - 1 \; and \; 1 <= j <= N - 2; S[i,j] -> S[i + 1, j] : 1 <= i <= N - 2 \; and \; 1 <= j <= N - 1\}$.

When we represent the dependency relationships between statement instances with blue arrows in the iteration space, we obtain a visualization like Figure 12. According to the basic dependency theorem , statement instances without dependency relationships can be executed in parallel. In the figure, there are no dependencies between the points within the green band (on the diagonal), so these points can be executed in parallel. However, we notice that the two-dimensional space has a basis of (i,j), corresponding to the i and j loops, which does not indicate parallelizable loops because the green band is not parallel to any axis.

To address this, Poly employs an affine transformation to change the basis (i,j) so that the green band becomes parallel to one of the axes of the space. This way, the corresponding loop can be executed in parallel. As a result, we can transform the space shown in Figure 12 into a form like that in Figure 13.

To put it differently, we can visualize the dependency relationships among statement instances as blue arrows within the iteration space, as shown in Figure 3. Based on the fundamental theorem of dependency, instances without dependencies can run concurrently. In the figure, all points within the green diagonal band have no dependency relationships and can be executed in parallel. However, the current (i, j) basis of the two-dimensional space does not allow for identifying parallel loops, as the green band is not parallel to any axis. Poly utilizes an affine transformation to modify the (i, j) basis, aligning the green band with an axis and enabling the corresponding loop to run in parallel. Thus, the iteration space shown in Figure 12 can be transformed into the representation in Figure 13.

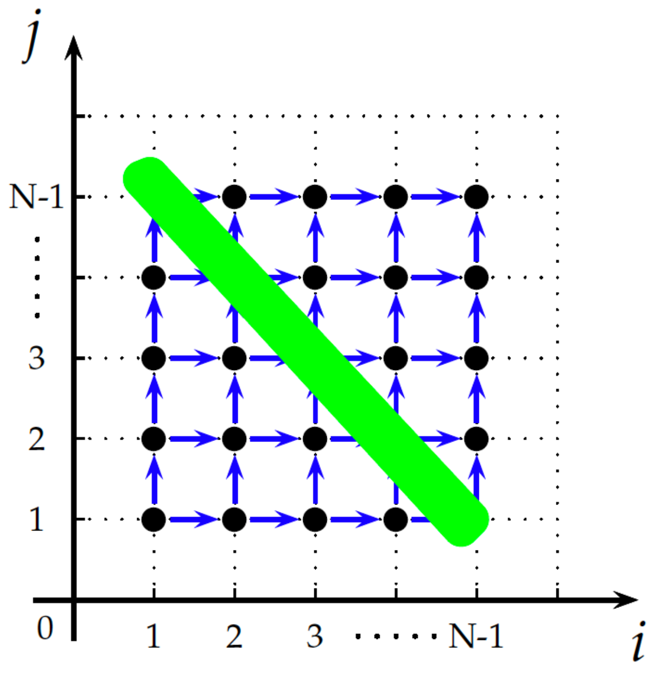


Figure 12 – loop before transform

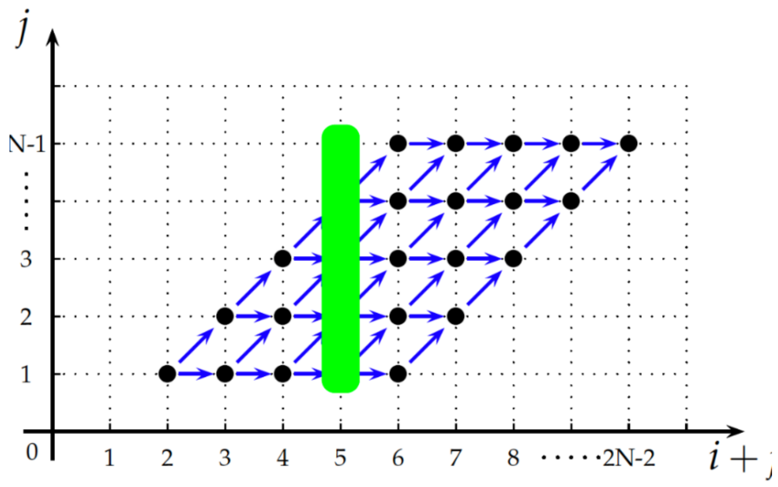


Figure 13 – loop after transform

Afterwards, the compiler will automatically generate code with good parallelism based on the dependency relationships identified by Poly.

3.1.3.2. Loop tiling in polyhedral optimization

In polyhedral optimization, loop tiling is an essential optimization technique. It divides the iteration space of nested loops into smaller sub-blocks, allowing calculations to be performed on smaller data sets. Loop tiling can improve locality, increase cache hit rates, reduce cache replacement, and optimize the utilization of multi-level caches. Moreover, loop tiling also contributes to enhancing parallelism, as computation tasks can be executed in parallel on smaller sub-blocks with the support of computing resources such as multi-core processors, vector instructions, or GPUs.

The most critical tiling/blocking (blocking) in deep learning applications, the purpose of block is to make full use of the cache on the acceleration chip. Through mathematical transformation, Poly can automatically realize more complex block shapes that are difficult to achieve manually. The implementation of block is very simple on Poly. You only need to modify the affine function in the band node to get the schedule tree corresponding to the block. As shown in Figure 15 is the scheduling tree after block in Figure 14.

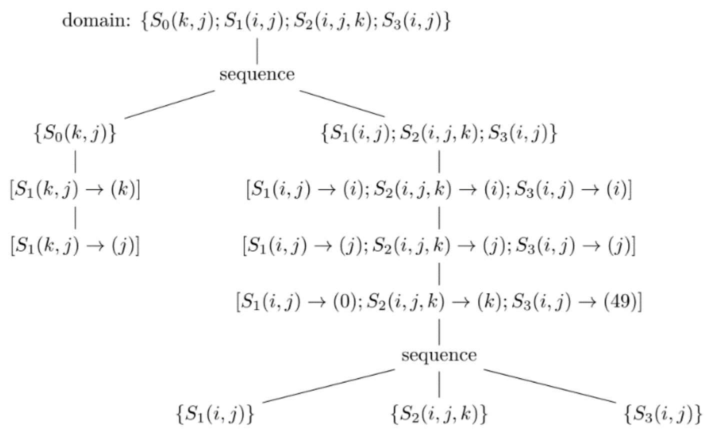

domain: $\{S_0(k,j); S_1(i,j); S_2(i,j,k); S_3(i,j)\}$
sequence
$\{S_0(k,j)\}$
$[S_1(k,j) \to (k)]$
$[S_1(k,j) \to (j)]$
$\{S_1(i,j); S_2(i,j,k); S_3(i,j)\}$
$[S_1(i,j) \to (i); S_2(i,j,k) \to (i); S_3(i,j) \to (i)]$
$[S_1(i,j) \to (j); S_2(i,j,k) \to (j); S_3(i,j) \to (j)]$
$[S_1(i,j) \to (0); S_2(i,j,k) \to (k); S_3(i,j) \to (49)]$
sequence
$\{S_1(i,j)\}$
$\{S_2(i,j,k)\}$
$\{S_3(i,j)\}$

Figure 14 – poly schedule structure

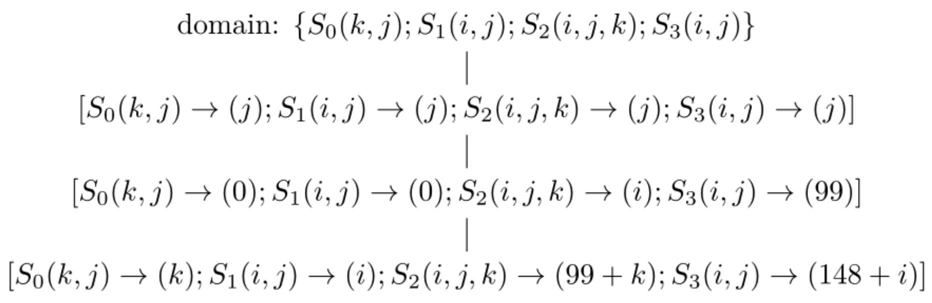

domain: $\{S_0(k,j); S_1(i,j); S_2(i,j,k); S_3(i,j)\}$
$[S_0(k,j) \to (j); S_1(i,j) \to (j); S_2(i,j,k) \to (j); S_3(i,j) \to (j)]$
$[S_0(k,j) \to (0); S_1(i,j) \to (0); S_2(i,j,k) \to (i); S_3(i,j) \to (99)]$
$[S_0(k,j) \to (k); S_1(i,j) \to (i); S_2(i,j,k) \to (99+k); S_3(i,j) \to (148+i)]$

Figure 15 – poly after tilling

3.2. Introduction to existing loop tiling optimization algorithms

Loop tiling is a compiler optimization technique that focuses on improving cache performance during program execution by breaking large loops into smaller sub-blocks. This approach offers various advantages, such as enhancing data access locality. By dividing data into smaller subsets, loop tiling increases the likelihood of the processor finding the data in the cache, reducing memory access latency.

Furthermore, loop tiling enables the computation process to be divided into smaller sub-blocks, making it easier to load and reuse data in the cache during loop execution. This significantly increases the cache hit rate and reduces cache misses, leading to improved program performance.

Additionally, loop tiling helps confine computation to smaller data sets, reducing data replacement in the cache. This means that the likelihood of data being replaced in the cache when executing sub-blocks is lower, further decreasing memory access latency.

Loop tiling also allows for optimized utilization of multi-level cache structures, which are common in modern computer systems. By adjusting the size of sub-blocks based on the capacity and access latency of different cache levels, better data reuse between multiple cache levels can be achieved.

Finally, loop tiling enhances parallelism by enabling computation to be performed on smaller sub-blocks. This opens up possibilities for utilizing computational resources such as multi-core processors, vector instructions, or GPUs. By further optimizing parallel strategies based on loop tiling, program execution performance can be improved.

The following new works are optimized around these points.

3.2.1. SICA

The loop tiling[1] approach is based on a combination of static and empirical analysis using machine learning techniques, integrating loop transformations with a fast, automatic, hardware-aware, and direct static TSS model. This method automatically adapts strategies for different application codes, facilitating vectorization and multi-core parallelism. Additionally, the approach employs an automatic parallelization and locality optimization tool called Pluto to compute optimal tile sizes, incorporating them as part of the loop tiling process. By doing so, cache misses can be minimized, and communication between different cores on multi-core processors can be reduced to the lowest extent possible.

3.2.2. TurboTiling

The primary contribution of this paper[2] is the introduction of an algorithm called TurboTiling, designed to optimize the performance of loop tiling code. Loop tiling is a common optimization technique that enhances temporal locality by dividing the problem domain into smaller chunks and repeatedly accessing data within them. However, loop tiling disrupts the data streaming access pattern, preventing the utilization of hardware prefetchers.

To address this issue and improve scalability, the TurboTiling algorithm employs a strategy that focuses on tiling for the last-level cache, taking advantage of hardware prefetchers and reducing off-chip traffic to memory. The algorithm also proposes a new tile size selection strategy to maximize data reuse within the last-level cache.

The paper further presents experimental results, validating the performance benefits of the TurboTiling algorithm on multi-core processors and comparing it to other existing algorithms. The findings demonstrate that TurboTiling significantly enhances the performance of loop tiling code and exhibits excellent adaptability across various types of applications.

3.3. Non-Vector Multiple Loop Tiling Algorithm

Although the aforementioned algorithms are among the more advanced loop tiling techniques currently available, their performance is notably effective only when dealing with vector integer multiples.

In general, when the innermost layer of nested loops has no cross-iteration dependencies, compiler auto-vectorization will automatically handle them. Standard loops can be straightforwardly tested without considering unaligned data access, as it does not affect vectorization. This paper focuses on non-standard loop blocks where the loo0p count is not an integer multiple of the vectorization width. In such cases, any form of tiling may lead to unaligned data access. Compared to other TSS algorithms, the proposed algorithm demonstrates highly efficient performance in handling these non-standard situations.

3.3.1. Maximum vectorization factor evaluation

For a program to be vectorizable, it must meet two requirements: first, the data access must be continuous, and second, the accessed data addresses must be aligned. Although loop tiling changes the loop boundaries of a program and subsequently affects the address alignment of the data accessed by vector registers, it does not impact the continuity of data access within the tiled blocks. As a result, our primary focus lies in addressing the alignment issue. In this context, we define the number of vectorizable elements as NUM_VEC, which represents the total count of data elements within all vectorizable blocks. A larger NUM_VEC value corresponds to greater vectorization benefits. To maximize vectorization gains, this paper proposes selecting a tile size factor for the vectorizable loop level that maximizes the number of vectorizable data elements within the loop body, ensuring the most significant possible vectorization advantages.

```
for(i=0;i<N;i++)
    for(j=0;j<N;j++)
        for(k=0;k<N;k++)
    S:    C[i][j]+=A[i][k]*B[k][j];
```

Figure 16 – loop example

```
for(iT=0;iT<N/I;iT++)                          //inter-tile
   for(jT=0;jT<N/J;jT++)
      for(kT=0;kT<N/K;kT++)
         for(i=I*iT;i<I*iT+I;i++)              //intra-tile
            for(k=K*kT;k<K*kT+K;k++)
               for(j=J*jT;j<J*jT+J;j++)
                  S:   C[i][j]+=A[i][k]*B[k][j];
```

Figure 17 - Blocked nested loops

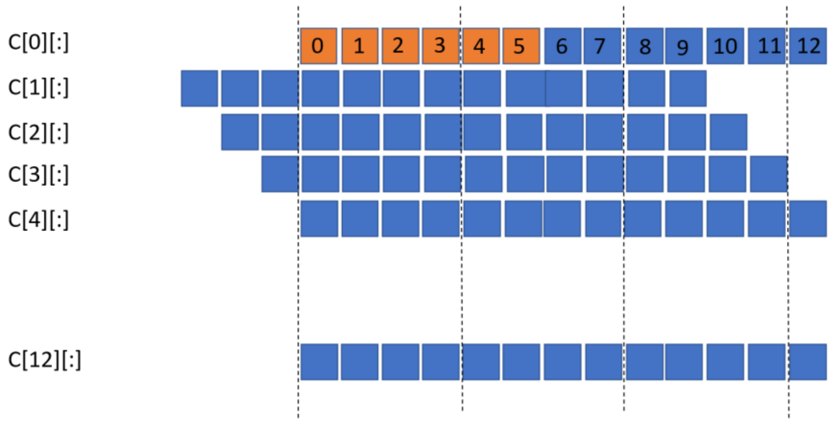


Figure 18 - addresses are misaligned

As depicted, when N equals 13 and the tile size is chosen as 6, data at positions 0, 1, 2, and 3 can be vectorized properly, while positions 4 and 5 cannot. For the second tile, $C[0][6:11]$, positions 6 and 7 are excluded, but positions 8 through 11 can be vectorized successfully. With each successful vectorization, the accumulated data count increases by 4. Eventually, the maximum vectorized data value, NUM_VEC, is determined. For instance, when the tile size is 6, NUM_VEC equals 104. By determining the maximum value of num_vec based on the array range N and the vectorization width, the optimal tile size for maximizing vectorization gain can be identified.

3.3.2. Locality and parallelism of loop tiling

3.3.2.1. Locality

Nevertheless, loop tiling must take into account both locality and parallelism. Therefore, after finding the maximum gain factor, these aspects must also be analyzed. To evaluate data locality, the reuse distance (RD) is commonly used as a metric. Considering the previously mentioned loop, the terms are represented as (I, J, K). For the array $B[k][j]$, the outermost layer i has to complete one iteration for array B to be reused. In this process, K distinct data points from array $A[i][k]$ are accessed. The reuse distance for array B, RD_B
is derived from the J distinct elements of array $C[i][j]$ and the $KJ - 1$ distinct elements of array $B[k][j]$, resulting in $RD_B = K + J + (K * J - 1)$.

Based on the literature and the previously discussed loop, we know that RD_B represents the longest reuse distance. When RD_B can be reused, the other two reuse distances can also be reused.

When the data reuse distance RD_s of the loop statement S within a tile is smaller than the size of a certain cache level, it indicates that all arrays in the statement S can be reused in that cache level. As a result, the entire working set of the tile can be mapped to that cache level to exploit the locality benefits within the tile.

When a cache stores both data and instructions, it is a common practice to allocate 75% of the cache capacity for data storage. This approach is based on the following considerations:

Balancing storage for data and instructions: In computer systems, both data and instructions need to be loaded from main memory into the cache for quick access by the CPU. To ensure that the cache provides sufficient space for both data and instructions, the capacity for data storage is limited to a portion of the cache capacity, in this case, 75%.

Avoiding cache pollution: In some situations, certain data or instructions may occupy a large portion of the cache, preventing other important data or instructions from being loaded into the cache. This scenario is referred to as cache pollution. Limiting the capacity for data storage to a portion of the cache capacity can help mitigate this issue.

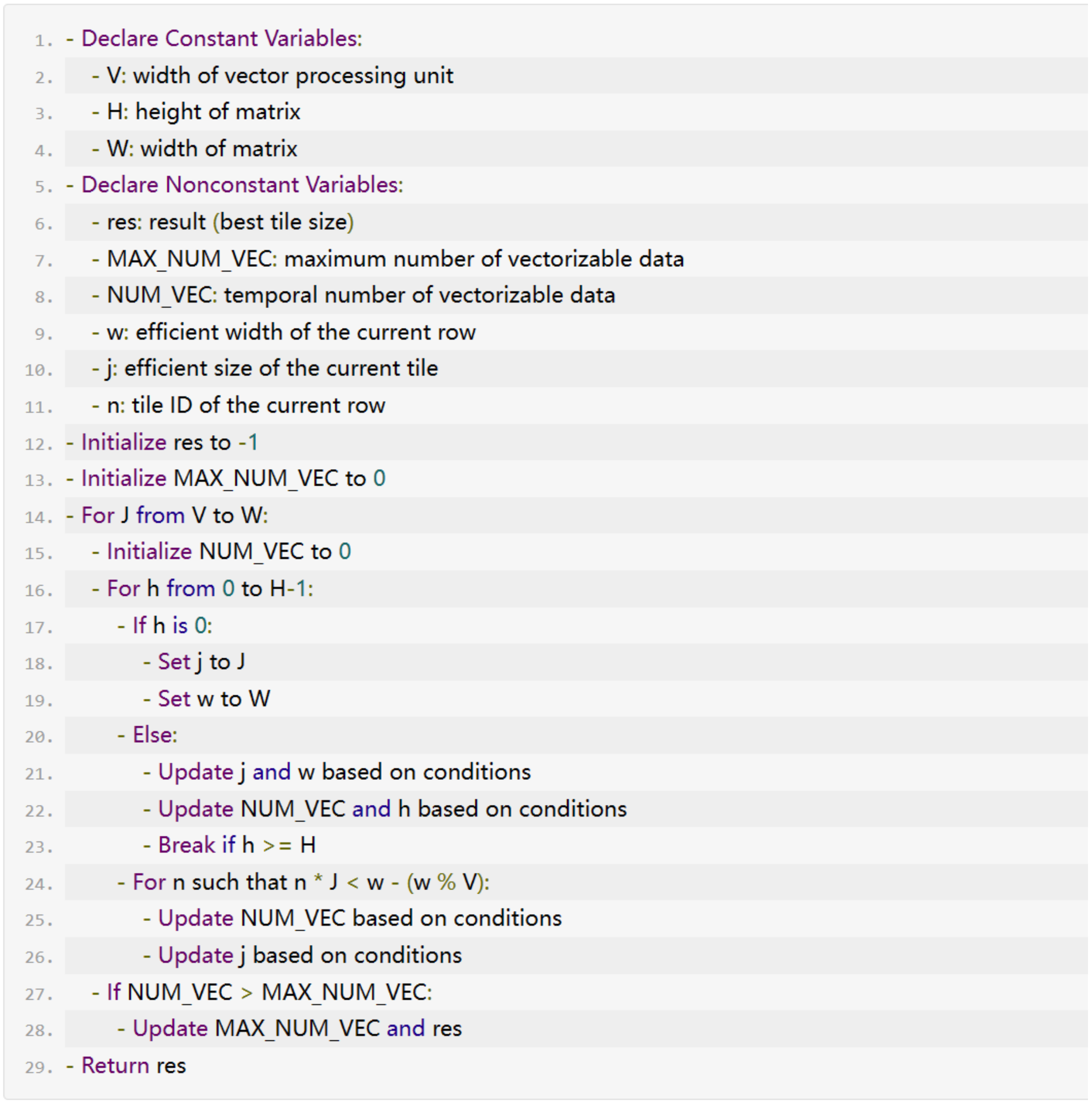
For shared caches in multi-core processors, the capacity available for data storage is set to 1/r of the cache capacity, where r represents the number of cores. This design principle aims to ensure that each core can fairly access resources from the shared cache, preventing performance degradation of other cores due to one core occupying excessive cache resources.

It is important to note that the 75% and 1/r ratios are empirical, and adjustments may be needed based on specific hardware and software requirements in actual applications. The choice of these ratios is driven by a balance between performance, cache utilization, and fairness.

3. Parallelism

To maintain good parallel granularity, load balancing across multiple cores is required. According to literature [8], when each core processes more than 2 blocks, the parallel gain of the program can be ensured. It helps to maintain load balancing by distributing the workload evenly across multiple cores, maximizing parallelism. It also reduces overhead associated with parallel processing, such as communication and synchronization, by amortizing it over a larger number of blocks. Lastly, it ensures better scalability as the algorithm can efficiently adapt to an increasing number of cores, leading to continuous performance improvements with the addition of more cores.

Function getBestTileFactor(V, H, W):



```
- Declare Constant Variables:
  - V: width of vector processing unit
  - H: height of matrix
  - W: width of matrix
- Declare Nonconstant Variables:
  - res: result (best tile size)
  - MAX_NUM_VEC: maximum number of vectorizable data
  - NUM_VEC: temporal number of vectorizable data
  - w: efficient width of the current row
  - j: efficient size of the current tile
  - n: tile ID of the current row
- Initialize res to -1
- Initialize MAX_NUM_VEC to 0
- For J from V to W:
  - Initialize NUM_VEC to 0
  - For h from 0 to H-1:
    - If h is 0:
      - Set j to J
      - Set w to W
    - Else:
      - Update j and w based on conditions
      - Update NUM_VEC and h based on conditions
      - Break if h >= H
    - For n such that n * J < w - (w % V):
      - Update NUM_VEC based on conditions
      - Update j based on conditions
  - If NUM_VEC > MAX_NUM_VEC:
    - Update MAX_NUM_VEC and res
- Return res
```

Figure 19 - Pseudocode

4. ALGORITHM PROGRAMMING

4.1. Introduction to Pluto

The PLuTo compiler is an automatic parallelization tool based on the polyhedral model. Its primary contributions lie in the introduction of an effective scheduling algorithm and the implementation of a software platform for polyhedral compilation. Numerous subsequent studies related to polyhedral compilation have built upon this scheduling algorithm, expanding implementation, experimentation, and testing on the platform. PLuTo has garnered over 700 citations in the niche field of polyhedral compilation. Today, this approach is widely employed in various high-performance computing domains, such as machine learning.

The primary objective of PLuTo is to transform a C program into a parallelized C program utilizing OpenMP. Subsequent practices have strengthened or extended support for SIMD/GPU/MPI/PluTo+/llvm-libpluto and more.

In 2018, PLDI awarded the paper with the Most Influential Paper Award, recognizing its core contributions to the practical application of polyhedral compilation, the inspiration it provided for subsequent research, and the enablement of new applications.

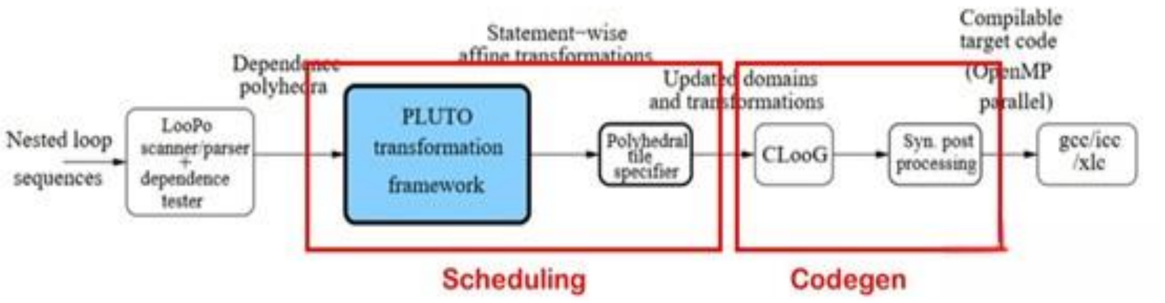

Nested loop sequences
LooPo scanner/parser + dependence tester
Dependence polyhedra
Statement-wise affine transformations
PLUTO transformation framework
Polyhedral tile specifier
Updated domains and transformations
CLooG
Syn. post processing
Compilable target code (OpenMP parallel)
gcc/icc /xlc
Scheduling
Codegen

Figure 20 – Pluto structure

Another reason for choosing PLuTo is its extensibility. Additionally, the code mentioned in the article was tested using this compiler, facilitating comparisons.

4.2. Test benchmark

In this study, the selected benchmark tests were derived from the PolyBench 3.2 suite, which is also utilized by SICA and TurboTiling. PolyBench is a comprehensive benchmark suite containing static control parts, designed to standardize the execution and evaluation of kernels often used in research publications. The key features of PolyBench include:

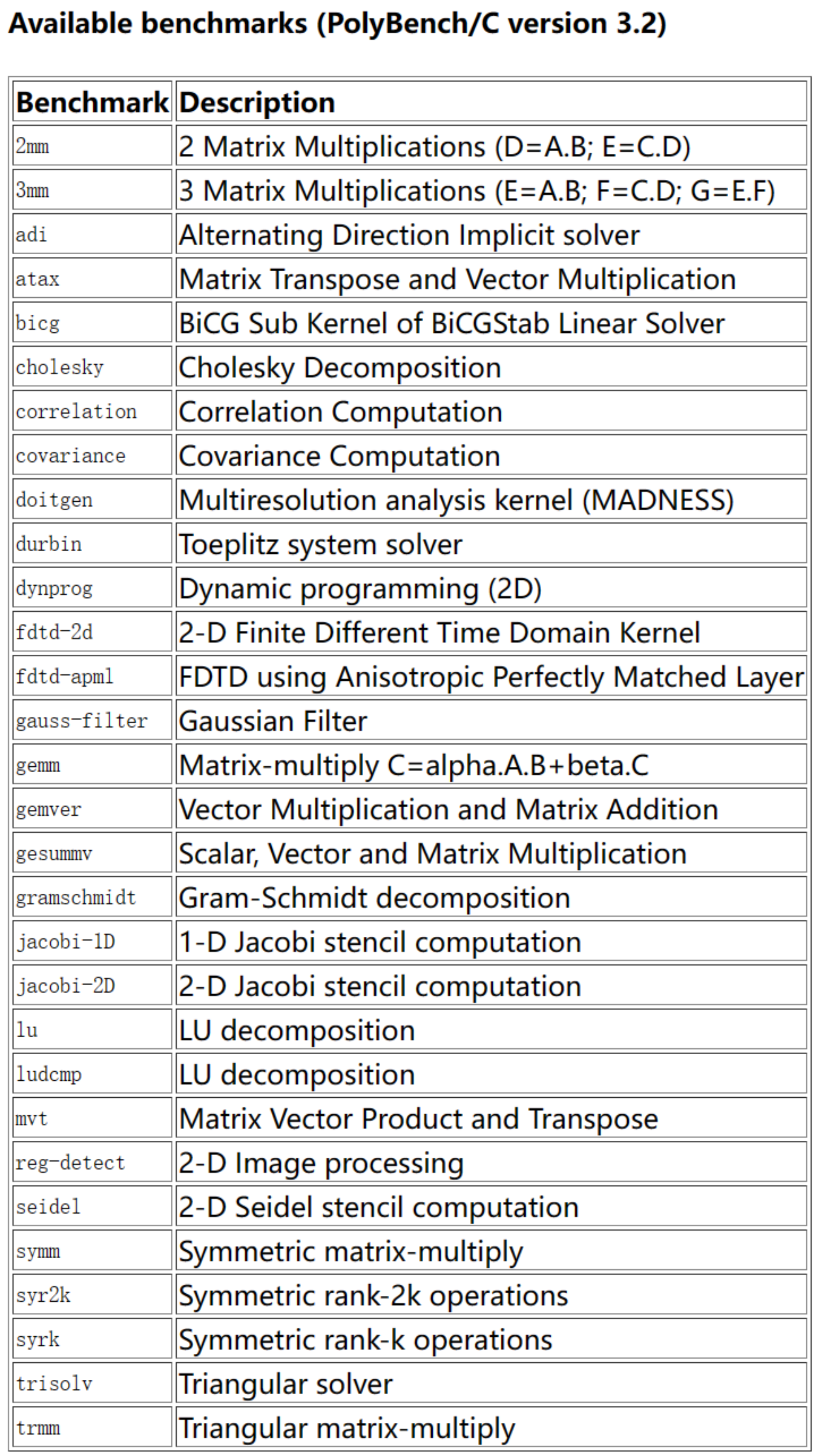
A single, compile-time tunable file for kernel instrumentation, which performs additional operations such as cache flushing prior to kernel execution and real-time scheduling to minimize OS interference.

Non-zero data initialization and live-out data output.

Syntactic constructs to prevent dead code elimination within the kernel.

General-purpose implementation using parametric loop bounds in the kernels.

Clear demarcation of kernels using #pragma scop and #pragma endscop delimiters.



Available benchmarks (PolyBench/C version 3.2)

Benchmark	Description
2mm	2 Matrix Multiplications (D=A.B; E=C.D)
3mm	3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
correlation	Correlation Computation
covariance	Covariance Computation
doitgen	Multiresolution analysis kernel (MADNESS)
durbin	Toeplitz system solver
dynprog	Dynamic programming (2D)
fdtd-2d	2-D Finite Different Time Domain Kernel
fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer
gauss-filter	Gaussian Filter
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition
mvt	Matrix Vector Product and Transpose
reg-detect	2-D Image processing
seidel	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k operations
syrk	Symmetric rank-k operations
trisolv	Triangular solver
trmm	Triangular matrix-multiply

Figure 21 – test bench example

In this study, 8 benchmark test programs capable of vectorization were chosen from the PLuTo benchmark suite, PLuToBench, and the polyhedral compilation optimization domain PolyBench/c 3.2 benchmark test set. Among these test programs, matmul, dsyrk, dsyr2k, lu, and trisolv involve calculations using double-precision floating-point numbers, while the test programs tmm, corcol, and covcol use single-precision floating-point numbers for calculations.

4.3. Program structure

PLuTo is a code transformation tool designed to optimize nested loops. Two years ago, it introduced a mature TSS (Tile Size Selection) interface, which significantly improved its flexibility compared to the previous default tile size of 32. Now, users can easily perform calculations based on the interface, and this thesis demonstrates the integration and application of the algorithm within PLuTo.

(1) This thesis's algorithm module is activated when there is a vectorizable loop level within the nested loop. PLuTo leverages the Candl and isl modules to extract various program features into the polyhedral model, including problem size, number of arrays, data types, array indices, and loop level order. The algorithm module uses this information to compute the optimal tile size. To enable access to the calculation results, a global structure named vec_tile_size is employed to store the results.

(2) Accessing the algorithm results is made possible through the $tile.sizes$ file, which PLuTo uses as the interface for updating the tile size. PLuTo reads the tile size factors for each loop level from the $tile.sizes$ file using the read_$tile_sizes()$ function. By modifying the $read\_tile\_sizes()$ function, PLuTo can automatically utilize the data

stored in the vec_tile_size structure for code tiling.

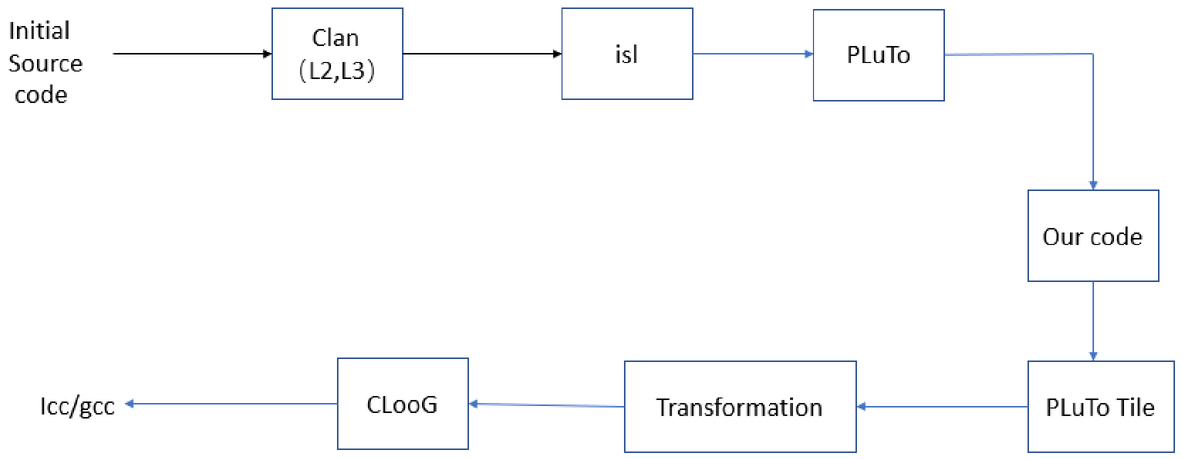


Figure 22 – code in Pluto

4.4. Test platform

Due to the long queue for new servers and the fact that my laptop has an AMD CPU with a chaotic scheduling system, I eventually compromised and conducted the experiments on an old Xeon CPU.

Hardware:

Table 1 - CPU parameters

CPU	Cores	L1	L2	L3	SIMD width
Intel Xeon E5-2685	1*8cores	8*32kb	8*256kb	16mb	256bit

Platform:

The experiments were conducted on a 64-bit server running Ubuntu 16.04 operating system. The gcc compiler was used with the O3, parallel, and openmp compile options. The version of PLuTo utilized in the experiments was pluto-0.11.4.

4.5. Analysis of experimental test results

Nested loops with different problem sizes have varying optimal tile sizes. In this study, we first tested a series of programs with ill-conditioned problem sizes, comparing our algorithm to the state-of-the-art TTS algorithm. We chose not to use the SICA algorithm because, as shown in the TTS algorithm paper, SICA is generally inferior to the TTS algorithm in most cases. To validate the effectiveness of our algorithm and exclude the influence of the number of threads, we executed all test programs using 8 threads. The figure below shows the average speedup of the tiled programs using both algorithms compared to the untiled original programs. As the problem size N increases, the advantage of our algorithm becomes more apparent. This is because as the problem size grows, the amount of data that can be vectorized also significantly increases, making the benefits of vectorization more pronounced.

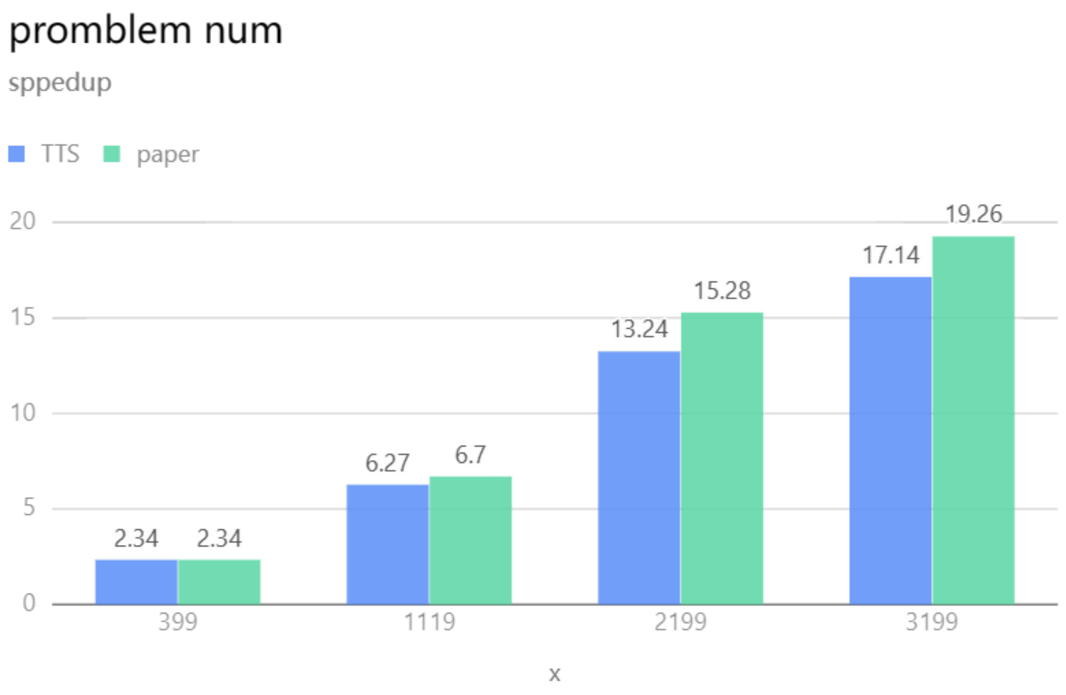


Figure 23 - Speedup ratios at different problem scales

In the eight test cases, the tile sizes derived from both algorithms for each benchmark test program are identical. This can be attributed to the high similarity in data reuse distances and working set sizes within the nested loops, even though the loop structures and loop bodies of these benchmark test programs differ from one another.

The following table shows the block size when the problem size is 3199, and the figure shows the speedup ratio of each program at this time

Table 2 - Tile size

	TTS	thesis
Tile size	40x16x3	41x40x3

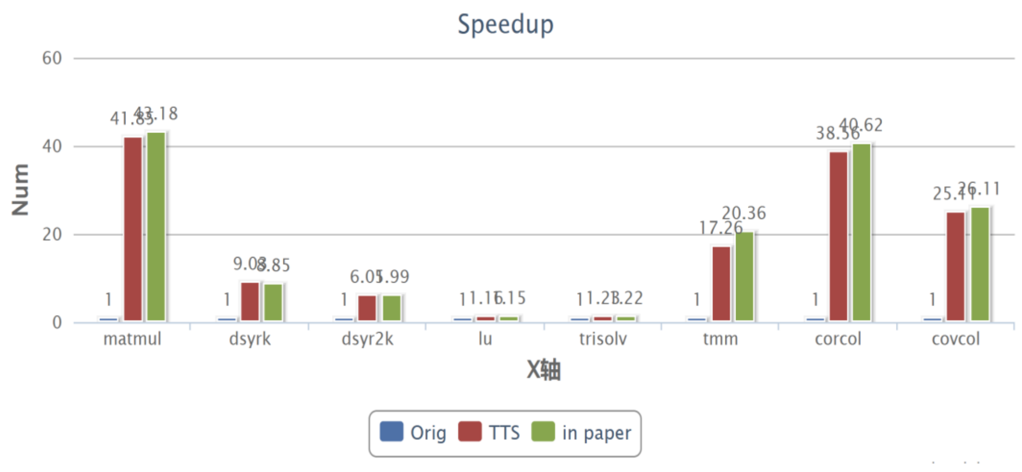


Figure 24 - Program speedup ratio

The acceleration performance of the test programs tmm, corcol and covcol after the algorithm is divided into blocks in the paper is better. Introduce the graph of this test set according to the table above. This is because the data of these three programs is 32-bit single-precision floating-point, while the others are 64-bit double, which leads to more unaligned data when using non-vector-aligned iterations. Thus slowing down performance. The algorithm in this paper can make certain optimizations for this situation.

4.6. Scalability analysis

In order to test its parallelism and scalability, we use export $OMP\_NUM\_THREADS = 1,4,8$ to test the performance of tiling under multiple thread conditions. The results show that all 8 algorithms have a certain degree of performance improvement. The algorithm in this thesis limits the block factor of the outermost loop through parallel granularity analysis, so that the number of loop blocks allocated to each core is greater than 2, ensuring load balancing among multiple cores.

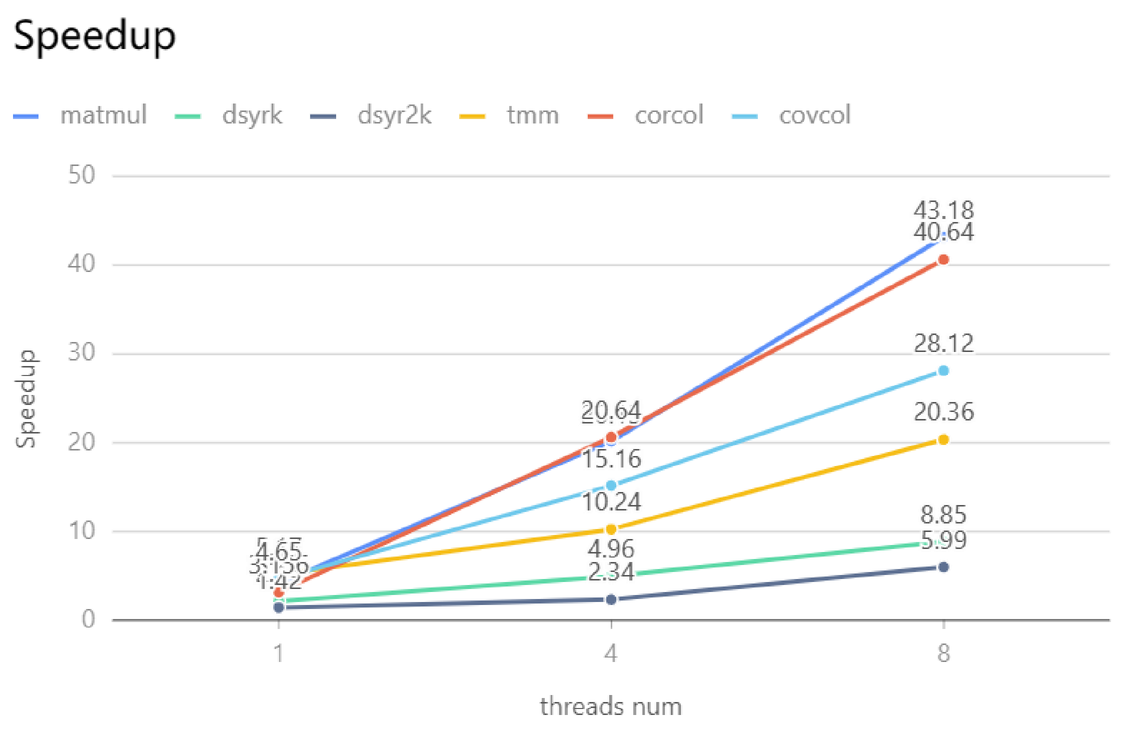


Figure 25 - Speedup ratio under multithreading

Table 3 - Problems under multithreading

Thread num	1	4	8
lu	1.56	1.57	1.49
trisolv	1.62	1.58	1.58

These two programs, LU decomposition and Triangular solver, exhibit good locality. However, due to data dependencies, they may cause false sharing when running in parallel. This can lead to data contention, which in turn prevents performance improvement during multi-threaded execution.

CONCLUSIONS

Our thesis focuses on addressing the problem of misaligned data accesses that can occur when using loop tiling to optimize nested loops. This issue can reduce the effectiveness of automatic vectorization techniques. To overcome this problem, we introduce a new algorithm that calculates the benefits of vectorization in advance to determine the best tile size. This algorithm is particularly useful for large-scale nested loops and helps to create loops with strong parallelism after tiling.

FUTURE PLAN

In light of the findings and discussions presented in this study, we propose the following future research directions to further enhance the optimization of nested loops and explore additional possibilities in this field:

Refining the proposed algorithm: Although the current algorithm has shown promising results, there is always room for improvement. Investigating alternative approaches to precompute vectorization benefits and incorporating advanced techniques for determining optimal tile sizes could lead to even better performance and parallelism.

Addressing false sharing and data races: As we discussed the potential issues of false sharing and data races in certain programs like LU decomposition and triangular solver, future work could focus on developing techniques to mitigate these problems and enhance parallelism in such cases.

Exploring heterogeneous computing environments: As modern computing systems increasingly adopt heterogeneous architectures, it would be valuable to extend the proposed algorithm to optimize nested loops on various hardware platforms, such as GPUs, FPGAs, or specialized accelerators.

Investigating loop fusion and other loop transformations: While this study focuses on loop tiling, there are other loop transformations that can be explored in conjunction with the proposed algorithm, such as loop fusion, loop distribution, or loop skewing. Investigating the synergy between these techniques and the current algorithm could lead to further optimization opportunities.

Real-world applications and case studies: To evaluate the practicality and effectiveness of the proposed algorithm, future work could focus on applying the algorithm to a broader range of real-world applications and case studies, assessing its impact on various domains such as scientific computing, machine learning, and computer vision.

By pursuing these research directions, we can continue to advance our understanding of nested loop optimization and contribute to the development of more efficient and parallelized software solutions in a wide array of applications.

ACKNOWLEDGEMENT

Finally, I would like to express my deepest gratitude to my supervisor, Alexey Paznikov, who has taught me invaluable knowledge about conducting research and guided me throughout this journey. I am also thankful to our class monitor for her dedication and timely communication, ensuring that important information was always shared promptly.

I am grateful for my fellow classmates, who have opened my eyes to a broader world and inspired me to aim higher. I extend my appreciation to our project leader, whose guidance and support have been indispensable in helping me overcome challenges and achieve my goals.

I would also like to thank Daria from the International Students Office for her unwavering assistance in addressing the many difficulties that international students may encounter, particularly in matters related to visas. Her support has been instrumental in making my educational experience a smooth and enriching one.

Last but not least, I am deeply thankful to all the Russian people who have helped me along the way, making my time in Russia a truly memorable and rewarding experience. Your kindness and generosity have left an indelible mark on my heart, and I am truly grateful for the opportunities and friendships that have come my way.

ADDITIONAL CHAPTER

Commercialization of the Research's Results

1. Summary

1.1. Description of the project

The project "Algorithms for Optimizing SIMD Instructions in Modern Architectures" aims to examine and improve compilation optimizations employed in current high-performance computing systems. This advanced technology is generally used within cloud servers, which are commonly used in modern industry. Our unique proposition is to lease this technology to the general public, and to users in other countries.

We have investigated the landscape of cloud computing service companies in Russia and combined this knowledge with our extensive experience in the field to develop a comprehensive business plan.

Our preliminary strategy involves an upfront investment of 1 million rubles, mainly designated for purchasing equipment and covering ongoing expenses during the initial months. We plan to operate for three years, after which we intend to grow our company and enhance the services provided to our clients. A revision of the business plan will be necessary at the end of the three-year period to adapt to the evolving market and company expansion.

1.2. Description of production

As a cloud computing company, our clients can access our services directly through the internet. Our sales are typically charged on a monthly basis, with discounts available for annual subscribers. There is no need to hire specialized maintenance personnel.

In the initial stages, we require our users to have a certain level of technical expertise. However, as time progresses, we will develop features such as control panels and automatic deployment to make our services increasingly user-friendly and accessible.

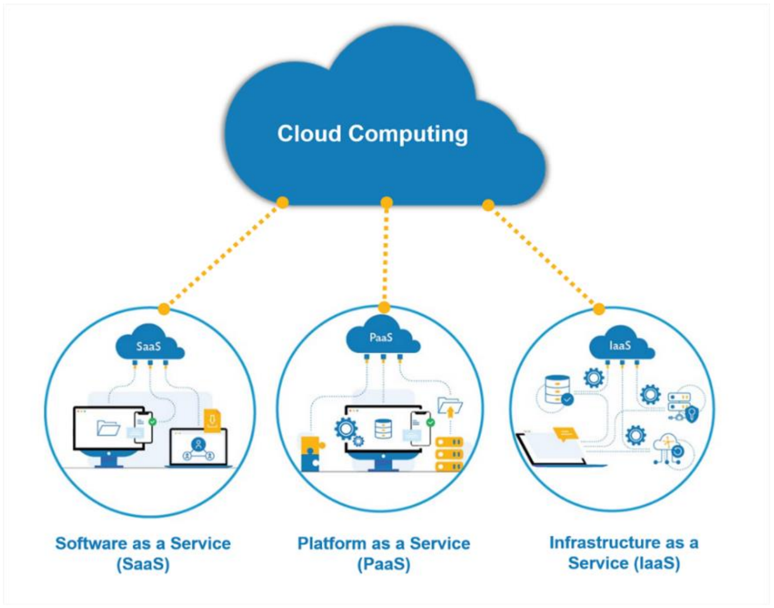

Cloud Computing
SaaS
PaaS
IaaS
Software as a Service (SaaS)
Platform as a Service (PaaS)
Infrastructure as a Service (IaaS)

Figure 26 – Cloud serviews

The table below is our chosen configuration

Table 4 - Server parameters

Item	Model	Cost
CPU	Intel Gold 8375c	69000*2
RAM	DDR4 32G	4000
DISK	WD 500GB	2500
Motherboard	Taian S7120 double-way	50000
Power	Quanhan 2U 800W	2500
Radiator	Pusai 2U LGA4189	2500
Tank	S208Y 2U	2500
Unit price		202000
Num		30
Total		6,060,000

Table 5 - Router parameters

Item	Cost
IE4320S-50S	94000

Table 6 - Resource

Software	Centos7
Price unit with VAT, rubles	4500
Revenue with VAT, rubles	2 000
resources	30 * 2 * 32 = 1920

1.3. Analysis of the market

Russia started embracing cloud services later than other countries, but this could work to its advantage as it sits on a massive wave of cloud computing. Although Russia currently holds only about 1% of the global cloud storage market share, research by iKS-Consulting shows that there is room for growth in the industry, with early predictions of a year-on-year growth of around 23% and an estimated value of 3.4 billion dollars by the end of 2022. The global demand for cloud computing has grown rapidly due to the impact of Covid, with Russia's growth rate reaching 24% in 2020, an astounding figure.

Initially, AWS was the most stable and largest market share holder among cloud service providers worldwide. However, after Telegram refused to submit data to the Russian government and was found to be using AWS services, the Russian government withdrew from AWS. Subsequently, due to a series of factors, Google's cloud services also left Russia. Since then, in addition to Russia's own Yandex Cloud and Mail.ru, smaller cloud service providers are also competing with each other, creating a blue ocean market.

After Telegram was blocked, it moved its servers to AWS and Google servers, thus bypassing the initial ban and continuing to provide services to Russian users. So Roskomnadzor, the Russian telecommunications regulator, was furious and banned 1.8 million IP addresses of AWS and Google cloud infrastructure in a fit of anger. Russian ISPs blocked 1,835,008 IPs at once. The move to ban the IP is in response to Telegram moving some of its infrastructure to AWS and Google cloud servers. So this incident also reflects the reason why Amazon does not have data centers in many places. The reasons are mostly related to cost and compliance requirements, including security and legal requirements. But this also embarrasses Russian customers who use cloud services, because Russian law also prohibits storing personal information of Russian customers outside Russia. But in this way, without the existence of AWS and Google Cloud, Chinese cloud service providers can see the Red Sea by investing in industrial parks and cooperating with local operators.

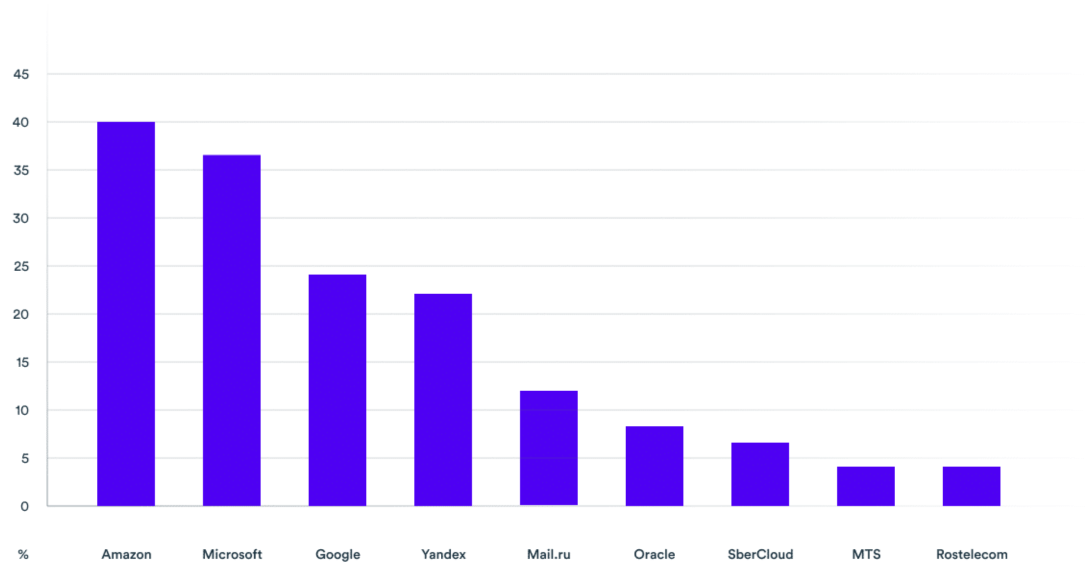


Figure 27 - Proportion of Russian cloud service companies before sanctions

Business leaders see many advantages in this landscape. Russia places significant emphasis on data privacy, and considering compliance is a major issue for international companies expanding their operations, entering a country with clear legal requirements actually makes the process of entering and using Russian cloud services relatively easier. With a high internet penetration rate, Russians have a strong demand for cloud services.

1.4. Competitor analysis

Current status of the Russian cloud computing market: According to a report released by the market research firm Canalys, in the first quarter of 2019, the total size of the Russian cloud computing market was US$850 million, a year-on-year increase of 25%. Among them, the cloud server market size was 290 million US dollars, a year-on-year increase of 29%. The Russian cloud computing market is in a period of rapid development, with an average annual growth rate of 30% to 40%. It is predicted that by 2018 the Russian cloud service market will grow 2.5 times to reach 500 million US dollars.

The development trend of the Russian cloud computing market: The development of the Russian cloud computing market is affected by many factors, including the political and economic environment, technological innovation, customer needs, laws and regulations, etc. On the one hand, the Russian government’s emphasis on information security and data sovereignty, as well as restrictions and supervision on foreign Internet companies, have brought challenges and uncertainties to the cloud computing market. For example, in 2018 the Russian government banned foreign communication applications such as Telegram and blocked 1.8 million IP addresses of AWS and Google cloud infrastructure. On the other hand, the Russian government is also promoting digital transformation and innovative development, providing opportunities and support for the cloud computing market. For example, the Russian government has formulated the "2024 National Digital Economy Project", which aims to improve the application and popularization of digital technology in the economic and social fields.

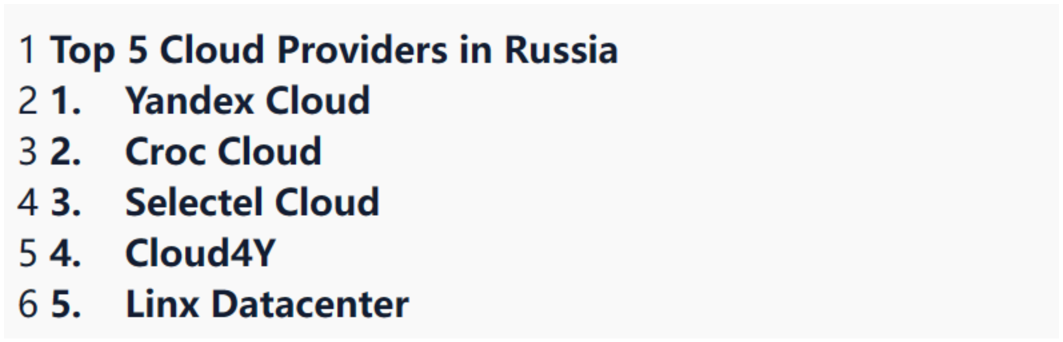
Major competitors in the Russian cloud computing market: The competitive landscape of the Russian cloud computing market is relatively complex, involving many domestic and foreign companies and brands. According to IDC data, in the first quarter of 2019, the top five players in the Russian public cloud service market were: Mail.ru Group (accounting for 18.6%), Yandex (accounting for 15.3%), Microsoft (accounting for 14.7%), Rostelecom (10.4%) and IBM (7.1%). Among them, Mail.ru Group and Yandex are local Internet giants in Russia with a strong user base and brand influence; Microsoft and IBM are internationally renowned IT companies with high market share and reputation in Russia; Rostelecom is Russia's largest One of the telecom operators in China, it has advantages in infrastructure and network coverage. In addition, there are some other competitors, such as Huawei, Alibaba Cloud, Google, etc.

Customer demand in the Russian cloud computing market can be analyzed from the following aspects:

The development of Russia's digital economy and the needs of digital transformation. The Russian government has formulated the "National Digital Economy Project 2024", which aims to enhance the application and popularization of digital technology in the economic and social fields. As an important foundation and driving force of the digital economy, cloud computing can provide efficient, flexible, secure and scalable services and solutions for Russia's key areas such as e-commerce, digital transportation, e-government, digital finance and "smart cities".

The needs of Russian businesses and social organizations. Russian business enterprises and social organizations use cloud computing technology for the following purposes: reduce costs, improve efficiency, enhance competitiveness, support innovation and adapt to changes1. According to the survey by J'son & Partners, the main reasons for Russian enterprises to use cloud services in 2019 are: saving money (48%), improving business agility (46%), simplifying IT management (38%), improving data security (36%), supporting remote work (32%), etc.

The needs of Russian personal and home users. The main reasons for Russian individuals and family users to use cloud services are: to enjoy more functions and services, to obtain better user experience, to protect personal data and privacy, to meet individual and diverse needs, etc. 1 . According to a survey by J'son & Partners, the main types of cloud services used by Russian individual users in 2019 are: online storage (67%), online games (54%), online videos (51%), online music (49%) , Online office (47%), etc.



Top 5 Cloud Providers in Russia

1. **Yandex Cloud**
2. **Croc Cloud**
3. **Selectel Cloud**
4. **Cloud4Y**
5. **Linx Datacenter**

Figure 28 – Top 5 in Russia

1.5. SWOT Analysis

Strengths:

Our competitive advantage stems from the exceptional talent of our workforce. We have assembled a highly skilled and creative team of cloud computing experts who are results-driven and possess outstanding qualifications and experience in various niche areas of cloud computing and related industries. In addition to the synergy within our carefully selected team, our services will be measurable, results-oriented, and guided by international best practices in the industry.

Weaknesses:

As a new cloud computing company in St. Petersburg, our organization may take some time to penetrate the market and gain recognition.

Opportunities:

Due to Russia's local policies, foreign cloud service providers with high popularity cannot enter the Russian market. Local mature providers, such as Yandex and Mail.ru, have not fully captured the market share, giving us an opportunity to enter the market.

Threats:

Inadequate Internet infrastructure. Russia's fixed and mobile network coverage, broadband access price and quality, and the size and distribution of data centers are far from developed countries such as Europe and the United States, limiting the availability and reliability of cloud services1. In addition, Russia lacks high-speed, low-cost, and secure cross-border network connectivity, which affects the international competitiveness and cooperation opportunities of cloud services.

Lagging and imperfect laws and regulations. At present, there are no laws and regulations specific to cloud computing in Russia, resulting in legal blind spots and risks in terms of rights and obligations, division of responsibilities, and dispute resolution between cloud service providers and users1. At the same time, Russia's laws and policies on data protection, data sovereignty, and data localization are also strict, limiting the development space and innovation potential of cloud services2.

Information security and data privacy concerns. Cloud computing technology involves a large amount of data storage, transmission, processing and sharing, which brings new challenges and threats to information security and data privacy. Russian users have low levels of trust in cloud service providers and fear that their data will be leaked, tampered with, damaged or misused. In addition, Russia is also facing cyber attacks and espionage activities from hackers, terrorists, foreign governments, etc., threatening national security and social stability.

So it is a wise investment to do cloud computing in Russia, although its risks are high.

2. Sales and Marketing

2.1. Sales plan

In the early stages, our primary focus will be on increasing brand awareness, offering our services at prices lower than the market average. As the years go by, our reputation will improve, our expertise will grow, and the overall experience of our cloud computing services will continually improve. Consequently, we will raise our prices, and as customers become more reliant on our services, they will continue to subscribe. While our costs may increase, our net profit will ultimately grow.

Table 7 – Sales plan

	First year	Second year	Third year	Total
Expected sales, units	8000	12000	16000	36000
Price with VAT, k rubles	13000	18000	20000	51000
Net revenue (excluding VAT)	6000	12000	13000	31000

2.2. Trade policy

Our team members have a certain level of reputation within the open-source community, generating a celebrity effect. Therefore, they can create promotional videos on video-sharing platforms, share on programming websites, and promote our products on school forums. As a result, we do not need to invest in advertising. In the beginning, our sales may be low, but due to the virality of the open-source community and word-of-mouth among students, and most importantly, our competitive pricing, our initial profits may not be impressive. However, after a few months, our products will quickly gain traction through word-of-mouth, leading to an increasing number of customers.

2.3. Sales policy and activities

Sales Policy: Pricing Strategy: We will offer competitive pricing for our cloud server services, offering monthly subscription plans and discounts for annual subscriptions. As our reputation grows and our services become more mature, we may gradually increase our prices while still remaining competitive in the market. Customer Segmentation: Our target customers will include small and medium-sized enterprises, start-ups, educational institutions, and projects of individual developers who need cloud server services. Customer Support: We will provide excellent customer support through various channels such as email, phone and live chat to ensure our customers have a seamless experience. We will also maintain a comprehensive knowledge base and FAQs to assist customers with common problems. Sales Activities: Online Presentation: We will create a user-friendly website to showcase our cloud server services, pricing plans and customer testimonials. We will also maintain a strong presence on social media platforms to engage with potential clients and share updates about our services. Content Marketing: Our team members have a strong reputation in the open source community and they will create educational content in the form of blog posts, videos and webinars to demonstrate the benefits of our cloud server services and attract potential customers.

Networking and Partnerships: We will actively participate in industry events, conferences and seminars to network with potential clients and partners. We will also explore strategic partnerships with complementary businesses and service providers to expand our market reach. Referral Program: We will implement a referral program to reward existing customers for referring new customers to our cloud server services. This will encourage word of mouth marketing and help us expand our customer base. With these sales policies and activities, our company will work in a studio in the outer suburbs of St. Petersburg. Keeping our overhead costs low while still providing excellent service to our clients.

3. Production Plan

Aside from the funds we allocate for purchasing servers, we will also incur one-time expenses for computer equipment and office furnishings. After these initial costs, our ongoing expenses will be periodic in nature, including utilities such as water and electricity, employee salaries, rent, and internet service fees.

Table 8 - One-time payment

	all
device	6,060,000
Office computer	150,000
office furnishings	400,000
All	6,475,000

Table 9 - Recurring payments

Cost	Year(1,2,3)			Three year
Rent	400000	450000	500000	1350000
Salary	3650000	4000000	4300000	1,195,0000
Water and Elec	2500000	2500000	2500000	7500000
All				20,800,000

Considering the loss and depreciation of hardware every year, based on the annual loss of 20%, the three-year loss rate is 20+16+12.8=48.8

Table 10 - Attrition charges

	Yearly Deprecation 20%	
	Cost	Deprecated amount in 5 years
Servers and Routers	6,060,000	2,957,280
Office Computer	150 000	73,200

Ultimately, we calculate that the total consumption over three years will be 30,305,480 rubles.

4. Financial plan

A financial plan outlines when and how much money will be spent and earned through the project. There are some fixed costs associated with the project, such as the project manager's salary and server rental fees, which remain constant from month to month.

During the first month, developers will be responsible for implementing features, testing applications, and setting up servers. After that, the service staff will receive the customers and the developers will be responsible for developing new functions and maintaining the server.

Table 11 - Employee salaries

Staff	Numbers	Salary
Customer service representative	2	45% of every year's salary
Technical supporter	2	55% of every year's salary

The company pays attention to equality, and the salary is actually far above the average salary, so there is not much difference in the salary of service personnel and

Table 12 - Changes in the number of employees and salaries

technical personnel, and it increases year by year with your working years

Years	1	2	3
Sellers/Administrator	2	2	2
Technical support	2	2	2
Total staff required	4	4	4
Total Salary per year	3650000	4000000	4300000

Year	I	II	III
Cash balance in the beginning	10000	9,707.2	29237
Service	24,000	27 645	35 325
Cash	19 584	33 712	40 336

Table 13 - Profit and loss plan without VAT

Inflow			
Profit	12060	18200	20540
Front-desk Laptop/Computer	150	0	0
Server and router	5 900	0	0
Electricity	2500	2500	2500
Salary(basic)	3650	4000	4200
Office rent	600	600	600
VAT	4000	6885	8500
Total outflows	16800	13985	15,800
Net cash flow			

ROI=(12060+18200+20540)/3/10000=1.693

NPV=3.15

IRR = 18.45%

It shows that the return on investment is good. we can invest.

Risk analysis

1. Legal and regulatory risks: Russia has strict data localization requirements, mandating that all personal data collected and processed within Russia must be stored on servers located within the country1. This can increase operational costs and complexity for some international cloud service providers and may also affect data security and accessibility2. Additionally, Russia has other laws and regulations, such as the Federal Law on Information, Information Technologies, and Information Protection, the Federal Law on Telecommunications, and the Federal Law on Personal Data, which impose certain restrictions and standards on the provision and use of cloud services3.

2. Market competition risk: The Russian cloud computing market is highly concentrated, with the top five providers accounting for over 80% of the market share. Russian domestic companies, such as Mail.ru Group, Yandex, MTS, and Rostelecom, primarily provide SaaS and IaaS services and dominate the market. These companies have strong brand influence, customer loyalty, government support, and localization advantages, which can pose significant competitive pressure for new entrants or small businesses.

3. Technological innovation risk: Cloud computing technology is a rapidly evolving and changing field that requires continuous research and innovation to adapt to customer needs and market changes. However, Russia lags behind developed countries such as the US and Europe in cloud computing technology and lacks independent research and innovation capabilities, mainly relying on the introduction and adaptation of foreign technology. This may lead to Russian cloud service providers lacking competitiveness and differentiation advantages in technology and may expose them to risks related to intellectual property and security.

Conclusion

In conclusion, investing in cloud computing in Russia appears to be a promising opportunity, considering the ROI is greater than 1 and is estimated to be around 1.6. This suggests that the potential returns on investment are substantial and could lead to significant profits. However, it is essential to carefully assess the risks and challenges associated with the legal and regulatory environment, market competition, and technological innovation. By strategically navigating these risks and capitalizing on the growing demand for cloud services in Russia, investors can potentially achieve a strong return on their investment in the country's cloud computing sector.

Reference for businesses plan

1. "Home - | TOP500", Top500.org, 2023. [Online]. Available: https://www.top500.org/. [Accessed: 10- May- 2023]
2. 2023. [Online].Available: https://incountry.com/blog/global-clouds-and-cloud-providers-in-russia/. [Accessed: 10 – May - 2023]
    a. [3]2022. [Online]. Available: https://sbercloud.ru/ru. [Accessed: 10- May-
3. 2023]

BIBLIOGRAPHY

1. Feld D，Soddemann T，Jünger M， et al.Hardware-awareautomatic code-transformation to support compilers in exploiting the multi-level parallel potential of modern CPUS[C] Proceedings of the 2015 International Work-shop on Code Optimisation for Multi and Many Cores，2015：1-10.

2. Mehta S， Beeraka G， Yew P C.Tile size selection revis-ited[J].ACM Transactions on Architecture & Code Optimization，2013，10（4）：35.

3. Mehta S，Garg R，Trivedi N，et al.TurboTiling：leveraging prefetching to boost performance of tiled codes[C] Proceedings of the 2016 International Conference on Supercomputing，2016：1-12.

4. T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In Proc. of the 2000 Int. Conf. on Parallel Architectures and Compilation Techniques, PACT '00, pages 237–246, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0622-4.

5. https://www.csa.iisc.ac.in/~udayb/slides/uday-polyhedral-opt.pdf

6. Elango, Venmugil, NormRubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover."Diesel: DSL for linear algebra and neural net computations on GPUs."In Proceedings of the 2nd ACM SIGPLAN International Workshop on MachineLearning and Programming Languages (MAPL), pp. 42-51. ACM, 2018.

7. A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multilevel tiling of imperfectly nested loops. In Proc. of the 23rd Int. Conf. on Supercomputing, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-498-0.

8. Liu Song, Wu Weiguo, Zhao Bo, Jiang Qing. Loop Tiling for Optimization of Locality and Parallelism[J]. Journal of Computer Research and Development, 2015,

52(5): 1160-1176.

9. Bondhugula U， Hartono A， Ramanujam J， et al.A practical automatic polyhedral parallelizer and locality optimizer[C]Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008：101-113.

10. Levine D., Callahan D., Dongarra J. A Comparative Study of Automatic Vectorizing Compilers // Journal of Parallel Computing. 1991. Vol. 17. pp. 1223–1244.

11. Maleki S., Gao Ya. Garzarán M.J., Wong T., Padua D.A. An Evaluation of Vectorizing Compilers // Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques (PACT-11), 2011. pp. 372–382.

12. Extended Test Suite for Vectorizing Compilers. URL: http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz