UDC 004.8 : 004.94 https://doi.org/10.20535/2786-8729.6.2025.331193

METHOD FOR SOFTWARE PIPELINING ON GRAPHICAL PROCESSING UNITS

Artemii Vinokurov*
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
https://orcid.org/0009-0006-7861-5138

Anatoliy Sergiyenko
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
https://orcid.org/0000-0001-5965-1789

*Corresponding author: a.vinokurov@kpi.ua

Graphics Processing Units (GPUs) play a significant role in high-end computations, including artificial intelligence. However, the GPU hardware is often underloaded. This forces an increased volume of GPU hardware to maintain a high throughput for task execution. The low loading of the GPU resources remains an actual problem and it needs to be solved now. Therefore, it is essential to seek methods that enhance GPU loading.

The research object is computational processes in modern processors, especially in GPUs. The purpose of this study is to review the software pipelining approach, its advantages and disadvantages, the techniques that can be used in it, including both instruction-level and decoupled versions, and to assess the effectiveness of this approach for the GPU.

To satisfy the requirements, different analysis methods were used. First, the architectural requirements to apply software pipelining were reviewed. Second, the original formulation and historical development of the approach were examined. Third, different levels of parallelisation to implement software pipelining were explored. Finally, *C*-slowing was proposed as an optimisation technique to overcome the adversities of the underutilisation of computational resources.

The research has revealed the abundance of proper software pipelining for GPU implementations. Whereas existing works review the possibilities of this technique, they are often overlooked in contrast to simpler multi-threading techniques. However, investigated researchers have defined the crucial limiting factor to computational resources as a constraint by memory overloading, specifically the pipelining registers. To address this, the *C*-slowing approach was suggested and theoretically evaluated. It demonstrated a possible increase of over 30% in GPU loading for the analysed algorithm, proving its applicability.

In conclusion, the software pipelining approach shows decent potential to optimise GPU algorithms, requiring further investigation. *C*-slowing could be utilised to handle the problem of underutilisation of computation.

Keywords: software pipelining, Graphic Processing Unit, C-slowing, retiming, synchronous dataflow.

1. Introduction

Graphics Processing Units (GPUs) play a significant role in high-end computations, including artificial intelligence. However, the GPU hardware is often underloaded. This fact forces an increased volume of GPU hardware to maintain a high throughput for task execution. Therefore, it is essential to seek methods that enhance GPU loading.

Although applying the pipelining approach utilizes the various resources of the highly pipelined architecture of the GPU, the pressure on one particular type of resource is higher, thus bottlenecking the overall performance of the program. The pipeline buffer registers' usage becomes a crucial part of the optimisation process as the problem of memory storage and loading between

the stages creates delays [1]. Registers handle data storage and instantly load every stage transition, creating a register overflow. This problem is often addressed to the scientists, and efforts are made to solve it, including those in [2], where technique redirects register spilling from the device memory to the faster shared memory. Another idea is to use the resources of the idle GPU cores, which is also viable [3].

Nevertheless, the mentioned proposals neglect the idea of register usage optimisation, and other resources are applied to compensate for it instead. Despite the visible improvements, the proposed solutions could meet limitations from different types of resources, such as under-utilising the registers due to a short circuit length.

The low loading of the GPU resources remains an actual problem and it needs to be solved now.

This paper overviews the software pipelining approach and programming methods based on it. Their analysis helped to propose a new method of *C*-slow retiming as a method of software pipelining which helps to program the parallel tasks in GPU and other architectures, and therefore, is able to increase the GPU throughput.

2. Literature review and problem statement

2.1. Software pipelining

Software pipelining is based on instruction execution parallelisation to increase single-core processor performance. The RISC architecture instructions provide the implementation of data reading, their calculation, and result storing at very different points in time. Memory access instructions are further divided into memory loading and storing. Software pipelining allows three kinds of instructions overlap: loading, computing, and storing. This feature is the most beneficial for sequential tasks, including loops. Every loop goes through several cycles with the same structure: load data, wait for the loading, compute data, wait for the computation, store data, wait for the storing, increment the loop counter, and check the exit condition. Therefore, every loop cycle could be pipelined and executed without waiting: load data, compute and cycle control, and store data. Eliminating this type of latency made software pipelining one of the most successful instruction-level parallelism techniques for its time and is used to this day [4].

An example of a loop without software pipelining optimisation is shown in Figure 1. The task performs a sum of two arrays element-by-element, noted A and B. The resulting array is stored in array C. The code is written using a microprocessor with AArch64 assembly language, which is typical for ARM architecture as a descendant of RISC. The data allocation section, which includes data allocation and wrapper syntax, is omitted. The examined block starts with the `_start` section, which loads the addresses of the arrays used and initiates loop-carrying registers. The loop starts by loading the current index input elements and incrementing the pointers. It then proceeds with the computational operation, stores the result, and handles the loop.

```
_start:
     ldr  x0, =A              // Load base address of A
     ldr  x1, =B              // Load base address of B
     ldr  x2, =C              // Load base address of C
     mov  x3, #5              // Loop counter
loop:
     ldr  w4, [x0], #4        // Load A[i], inc. x0 by 4
     ldr  w5, [x1], #4        // Load B[i], inc. x1 by 4
     add  w6, w4, w5          // Compute sum of A[i] and B[i]
     str  w6, [x2], #4        // Store sum in C[i], inc.
     subs x3, x3, #1          // Decrement loop counter
     bne  loop                // If not zero, repeat
```

Fig. 1: Loop without software pipelining

Given loop implementation is typical for assembly language programming or high-level language compilation results.. Figure 2 shows an example of code using the software pipelining technique. The example starts initiating variables and setting up counter registers. However, that is not the only list of operations done before the loop begins; it includes the pre-loading instructions for the first elements of the input arrays. Therefore, the computational instruction does not need to wait for the input data to load but proceeds with the next iteration's input value loading. The next iteration starts with the input data loaded during the previous one, and the cycle continues. Utilising simultaneous logic is beneficial for performance, making software pipelining essential for effective programs.

```
_start:
     ldr  x0, =A            // Load base address of A
     ldr  x1, =B            // Load base address of B
     ldr  x2, =C            // Load base address of C
     mov  x3, #5            // Loop
     ldr  w4, [x0], #4      // Preload A[0]
     ldr  w5, [x1], #4      // Preload B[0]
     subs x3, x3, #1        // Decrement counter
     beq  last_iteration
loop_pipelined:
     ldr  w7, [x0], #4      // Load A[i+1]
     ldr  w8, [x1], #4      // Load B[i+1]
     add  w6, w4, w5        // Compute C[i]=A[i]+B[i]
     str  w6, [x2], #4      // Store C[i]
     subs x3, x3, #1        // Decrement loop counter
     mov  w4, w7            // Move preloaded A[i+1]
     mov  w5, w8            // Move preloaded B[i+1]
     bne  loop_pipelined
last_iteration:
     add w6, w4, w5         // Compute last iteration
     str w6, [x2], #4       // Store last result
// Exit the program
     mov x8, #93 svc #0     // syscall: exit
```

Fig. 2: Loop with software pipelining

The following assumptions are made to measure the performance gains using the theoretical approach: both code pieces are run on the same RISC processing unit, with the arithmetic instructions, branching, and moving data between registers taking one cycle to complete, whereas memory operations require four cycles.

Table 1 presents a workflow of this loop with the software pipelining technique used. Rows represent instructions, whereas columns represent processor cycles. As shown, operations overlap beginning with the third cycle: the addition instruction is executed, whereas loading A and B is still not finished. The computation operation would need to wait for the loading operations to complete without a software pipelining application.

Therefore, due to the technique, the length of the loop body is eight cycles. In comparison, the length would be 12 cycles without the technique usage. Thus, speed up could be calculated as 12/8 = 1.5 times. The superscalar processing unit could improve this number if it performs branch prediction.

Software pipelining is a technique known from the 1970s as part of Fortran compilers on the time's supercomputers [5]. Despite that, the technique was mostly manually applied by the less powerful machine programmers. An engineer was required to reorder the instructions of the code to utilise memory and arithmetic logic parallelisation.

Table 1. Pipelining cycles

Cycle No	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Load A	A[i]								A[i+1]							
Load B		B[i]								B[i+1]						
ADD																
Store				C[i-1]								C[i]				
Reg. movings						A[i]	B[i]									
Loop counter.																
Branch																

As many programs were still written in Assembly languages, primitive instruction reordering was the only solution to increase the performance of the RISC machines. However, the more high-level languages, including C, were developing and spreading, hiding the possibility of manual optimisations behind the compiler. Therefore, a need for non-manual instruction retiming began to rise, requiring compilers to adopt software pipelining as a part of them [6].

Considering such a tendency, instruction-level parallelism became greatly utilised, allowing the loop iterations' executions to overlap, optimising all sorts of tasks processors would execute. On the other hand, the compiler's hidden optimisation greatly contributed to software pipelining being forgotten as a technique, at least on the instruction level. Another reason is the low popularity of Assembly language, so hardly any engineer is required to optimise the instructions manually.

With the modern level of processors, the software pipelining technique is not limited to the RISC architecture only. Similarly, an approach could be utilised on the thread level, opening it to another optimisation level utilising multi-core architectures, including the highly promising GPU. Its level of parallelisation could benefit from the software pipelining optimisation on the level of threads.

2.2. Decoupled Software Pipelining

Software pipelining increases the productivity of repetitive algorithms on the hardware level for a single core. However, the utilisation of a higher level of parallelisation became mandatory for practical algorithms. Considering this, software pipelining was adopted to the multi-threaded environment as a new approach, called Decoupled Software Pipelining (DSWP). DSWP allows dividing complex computational operations inside the loop to be executed on separate threads [7].

The general idea of the DSWP technique could be expressed as follows: it partitions the loop body into several parts called *stages*. A thread executes a single pipeline stage and communicates to neighbouring stages via a queue buffer (QB). Every stage receives dependent data from the previous stage and then sends dependent data to the following stage. The absence of dependence cycles is the main requirement for this kind of pipeline. Buffering stage communication in a queue ensures the pipeline stages are decoupled and insulated from stalls in other stages.

This technique implements the DOPIPE approach of parallelisation. It contrasts with the other tactics: DOALL and DOACROSS. The former is used in the conditions of a loop having steps fully independent from each other's result, which maximises performance, utilising all the available threads it requires. However, iteration dependencies are typical for the vast amount of application code. And even the code transformations cannot guarantee DOALL applicability [8]. The DOACROSS is applied to the loops with partial dependency. Despite having inter-step dependencies to some extent, it produces partially parallelised execution, resulting in performance gains. Opposing these approaches, DOPIPE is used for the entirely dependent loops. The dependency of actions inside the iteration makes another use case for the DOPIPE approach.

Figure 3 shows an example of code written in the C language, which handles inter-iteration dependencies with pipelining. The context of the function call, including the `main` function, is omitted. Firstly, the code checks the input values and then proceeds with the loop. The body of the loop consists of several function calls that depend on the result of one another, consequently in the given order: "*input* → *a* → *b* → *c* → *output*".

```
void calculate(const int* input, size_t size, int*
    output)
{
     if (input == NULL || output == NULL || sise == 0)
          return;
     for (size_t i = 0; i < size; i++) {
     int currentElement = input[i];
     int resultA = a(currentElement);
     int resultB = b(resultA);
     int resultC = c(resultB);
     output[i] = resultC;
    }
}
```

Fig. 3: Code for DSWP

Code dependencies for DSWP are commonly visually presented via a Program Dependence Graph. It is a directed graph containing two types of dependencies, which are data and control, illustrated by the edges. The nodes represent operations, including calculations, variable assignments, function calls, conditional operations, and loop continuation operations. Firstly, DSWP groups instructions that have a significant inter-dependence into Strongly Connected Components (SCCs), typically illustrated by a Directed Acyclic Graph of SCCs (DAGSCC). The most crucial limitation of the efficiency of the pipeline is the slowest stage, which stalls the whole execution. Thus, the second (dynamic) grouping phase is performed, forming clusters of nodes of the DAGSCC, where another one-to-one relation is created between a cluster and a thread.

DSWP performs load-balancing during clustering, so none of the threads overflows with work. Meanwhile, the cycle inspections should be run on the new graph so that no cluster cycles persist, preventing correctness and performance issues [8]. Another way to visually present DSWP is a table. Table 2 contains pipeline iteration details for the code in Figure 3. Its columns represent operations executed on its stage on a separate thread. Rows represent iterations, theoretically having equal execution time, in which every operation gets completed for the given input element. The row means a pipeline step whose direct dependency on the loop iteration is not required. Every cell contains the input element handled in the corresponding operation at the given iteration time; cells representing the same element have the same colouring.

Table 2: DSWP iterations

	Load	**Function A**	**Function B**	**Function C**	**Store**
Iteration 0	input[0]				
Iteration 1	input[1]	input[0]			
Iteration 2	input[2]	input[1]	input[0]		
Iteration 3	input[3]	input[2]	input[1]	input[0]	
Iteration 4	input[4]	input[3]	input[2]	input[1]	input[0]
Iteration 5	input[5]	input[4]	input[3]	input[2]	input[1]
Iteration 6		input[5]	input[4]	input[3]	input[2]
Iteration 7			input[5]	input[4]	input[3]
Iteration 8				input[5]	input[4]
Iteration 9					input[5]

The DOPIPE approach shows in this example, that every DSWP stage (thread) is specialised in its instruction, allowing optimisation for the corresponding task, including further parallelisation for complex tasks. Another advantage of pipelining is the static number of threads independent from the input data size, as the number of threads is defined by the operation amount, which is static for

the SIMD approach. In the given example, DSWP would handle any array with only five threads, whereas DOALL would require six threads to be optimal.

However, the problem of the DSWP is a potential imbalance of the stage execution time, as operations are different; as a result, the right load-balancing technique becomes significant for the overall performance. Regarding communication instructions, DSWP requires more instructions than DOACROSS [9]. In contrast, DSWP keeps all the threads active, unlike waiting states often encountered in DOACROSS implementations.

In conclusion, DSWP is an effective method of parallelisation, being a performance-efficient implementation of DOPIPE, which is advantageous to both DOALL and DOACROSS in terms of flexibility and keeping threads active if properly implemented. DSWP is an elegant adaptation of software pipelining to the processing units with several threads. With the increase in GPU performance and usage, DSWP could be used to utilise the resources.

2.3. Software Pipelining in GPU

GPU is currently widely used in different fields of science and engineering. Artificial intelligence and machine learning are among the most popular use cases. Modern GPUs with appropriate algorithms could handle various tasks, including scheduling issues [10] and showing high rates of operations per second [11]. Provided that GPU optimisation is crucial, this comes to the question of multi-threading algorithm optimisation because it relies on multi-core at its base, with modern GPUs having thousands of cores able to run thousands of threads in parallel.

GPU is divided into units called streaming multiprocessors in Nvidia terminology. Streaming multiprocessor essentially has a RISC-like architecture. It has a small set of instructions executed in one clock cycle in pipelined mode. The hundreds of registers provide effective local variable storing. Moreover, support for automatic array indexing and loop execution is also present. Regarding computational units, streaming multiprocessors comprise GPU cores grouped into warps, further grouped into thread blocks. Warps perform parallelising with a Single Instruction Multiple Data (SIMD) principle, where every thread performs the same operation on different data chunks. However, warps can perform different operations under the same streaming multiprocessor [12].

Streaming Multiprocessors comprise not only computational units but also memory of different levels. Fast-access memory is extensively used for pipelining, starting with the global memory allocated for the whole GPU and continuing with Streaming Multiprocessor registers. The shared memory exists for the less significant operations within a thread block, which is slower than the registers but still faster than global memory. Local memory is a thread-specific subdivision of global memory [13]. Delays in memory access in case of insufficient usage could limit the computational possibilities of the GPU. The frequent context switching between threads is supported by hardware register bank switching.

This level of parallelisation provides possibilities for both instruction-level pipelining and thread-level pipelining. GPU frameworks exist to access these resources, namely CUDA and OpenML conveniently. They opened the possibility for GPU to be effective [14] despite not providing all the solutions required [15], including instruction-level pipelining possibility, as the infrastructure provided by the manufacturers limit low-level parallelisation controls.

2.4. Pipelining Approaches

Despite the high effectiveness of adapting CPU-oriented software pipelining approaches to the GPU, many techniques have been developed for optimising pipelining on the GPU. Examining the VersaPipe [16], the library written specifically for the GPU, combining high-level and low-level control to reach up to 2.7x performance increase compared to the baseline. Another library is GOPIPE [17], which is built on top of VersaPipe. It encompasses automatic granularity adjustment for the given task, providing a 1.39x increase in performance over the previous solution.

Furthermore, there are several problems with the current GPU design that limit its potential, comprising GPU hardware being agnostic to the thread block specialisation from the software perspective, the transfers of memory between global memory and shared memory are coarse-

grained and lack of fine-grained memory access limits pipelining possibilities, the requirement of complex manual optimisation by the developer in CUDA, limiting warp specialisation usage. The article [18] presented a WASP system containing architecture and compiler utilising warp specialisation for the GPU. The compiler provides a 10% performance improvement, whereas its usage with an augmented WASP GPU provides 47% better performance than the baseline on average.

Nevertheless, the mentioned techniques are not based on software pipelining, which could improve performance in operations running on the GPU. Unfortunately, the existing scientific base lacks GPU-specific research, primarily focusing on general software pipelining adoption to superscalar architectures [19, 20].

Despite that, there is research regarding implementing software pipelining on the discrete GPU to optimise rasterization problem solving in [21]. However, it does not present task-independent optimisation for the GPUs. This work still showcases an efficient software pipelining utilisation.

2.5. StreamIt: a Model for Streaming Programs

The stream-oriented programs consist of a theoretically endless stream of data handled the same way for every chunk, fitting perfectly into the category where data parallelisation could be used effectively for optimisation. As the GPU is set to handle data parallelism, stream-oriented programs are one of the fields in which it can be used. An instrument was developed to effectively use parallel resources for streaming programs: a model and a corresponding programming language named StreamIt. Whereas exploiting it on the CPU with 16 cores gives up to 11.2 times faster than single-core performance [22], the GPU's potential takes another level.

StreamIt is a programming model where a user specifies data transformations with the help of the *filters* concept. Filters can pop any number of elements from the input, process the value, and then push it to the output. In some implementations, it could also have a peek operation that allows looking at the input value without removing it, unlike pop, which removes the element from the input. Such behaviour reminiscents the DSWP technique. The developer has several actions to take to write a program using StreamIt. Firstly, he needs to define the rate of pushed and popped amount of the elements. Secondly, choosing how to combine filters could be done in split-join, feedback loop, or pipelining. Nesting these components is possible, but the input and output limits are one for both [23]. Thirdly, the operations performed for every filter must be specified.

In general, StreamIt is intended only for the CPU systems. Adopting it to the GPU platform is a task that needs to be performed. The work [24] investigates the possibility of using this language to make GPU-oriented programs. Running StreamIt on the GPU required compiler manipulations, making the output a CUDA code. This process was followed by problems with the correct versioning and configuration so that each filter could get the necessary number of threads and register to work.

This method was tested in the framework with tasks like discrete cosine transform, and blocked matrix multiplication. The optimised software pipelining solution increased performance from 1.87x up to 36.83x compared to the single-threaded CPU, depending on the benchmark setup. This examples shows the perspectives of the software pipelining in GPU.

2.6. Modulo Scheduling technique

Software pipelining could effectively handle loops with small and large operation counts for their bodies. However, time is spent in either saturation or decay states for the latter, and the program execution becomes longer, limiting the method's effectiveness. A technique called modulo scheduling is used to address the problem. It takes more control over the operation ordering and optimises the pipelining schedule. Compared to plain software pipelining, modulo scheduling adds the possibility to increase predicted optimisation, reducing the impact of the unexpected latencies on the overall performance outcome [4].

One of the basic algorithmic components of modulo scheduling is loop unrolling. This technique transforms an original loop into a loop with a complex body, decreasing the number of

iterations by the given factor of *N*. This transformation is done by putting the *N* of original iterations in a single iteration of the complex body, resulting in more operations that could be pipelined.

The step of modulo scheduling is crucial as it rearranges the cycles for the operation before trying to organise them in a schedule. It builds a graph of data and logic dependencies of the loop. Then, the graphs of the consequent iterations connected due to unrolling are linked in a bigger graph divided into smaller *M* graphs. The *M* is for *modulo* in modulo scheduling. Therefore, iterations get retimed to have at most *M* operations in each, optimising the algorithm's execution for the required number of software pipelining stages.

Despite the length of the loop after unrolling, modulo scheduling performs the next step, which schedules operations inside the loop body. The essential parameter used in modulo scheduling is the initiation interval, which defines the number of cycles between the iterations. The lower bound of it is computed first and is called the Minimum Initiation Interval (MII). The lower the initiation interval, the better parallelisation can be utilised. The initiation interval is highly dependent on the *critical path*, which is the longest route in the graph that cannot be separated without violating dependencies.

The name Flat Modulo Scheduling (FMS) is used for the algorithm that passes through the loop only once. Nevertheless, as it is easy to implement, it leaves much potential for modulo scheduling performance optimisation underutilised. The more advanced version with MII, which has passed through the loop, is called Iterative modulo scheduling and provides more adjustments after the first partial schedule is defined.

Several more complex implementations of modulo scheduling include backtracking, increasing the initiation interval, slack modulo scheduling, and integrated register-sensitive iterative software pipelining improve an average ratio of initiation interval to MII [25].

The Hierarchical Modulo Scheduling (HMS) solves the significant problem of the number of operations to schedule, being better at extracting the potential of the multiple cores of the GPU. As every primitive operation of computation, branching, or assignment gets scheduled, the number of operations to process for the modulo scheduling becomes significant and requires grouping. It is performed by transforming the original graph into DAGSCC and scheduling the newly formed groups rather than individual operations. This technique is often used in DSWP. Nevertheless, the most performance-effective approach of IMS and HMS is still debatable [26].

2.7 CPU and GPU combination

The software pipelining in the systems of the homogeneous processing units (PUs) is considered above. However, modern systems have CPU and GPU on the same chip, sharing a memory pool and cache. Examples of such a system are AMD Ryzen 5 3400G, NVIDIA Tegra X1, and many others. Therefore, investigation of combined GPU-CPU systems is essential to estimate the software pipelining potential, especially on less powerful devices where optimisation is crucial. This type of architecture has separate first-level caches for each PU, and the level two or three cache is shared via the unified memory architecture. Thus, exchanging data between CPU and GPU could be done similarly to simply exchanging memory between different threads inside the single PU [27].

There are three main optimisation possibilities with promising prospects for the CPU-GPU heterogeneous systems. Firstly, fine-grained communication and DSWP approach can decrease the temporal distance between the consumer and the producer. Second, the amount of memory access contention can be limited through effective access modulation, optimising cache usage. Third, the number of underused PU cores that require data-independence detection and task delegation to the idle threads should be decreased.

The investigation of such possibilities [28] has shown a significant performance increase for some common algorithms. The combined systems have possibilities to increase performance: using the shared memory and DSWP approach has a 2.1x increase compared to the baseline, further parallelisation of the CPU and GPU computing gives 3.1x speedup, and utilising cache increases the number to the 4.4x level [29].

Software pipelining has proven to be a potent technique for instruction-level pipelining, later being reintroduced for the multi-core architecture in the decoupled form. However impactful software pipelining was for other architectures, it lacked productive implementations for the GPU. The efforts to use it for rasterization gave a plausible result, but it has not received enough attention and development. Another promising research field is applying software pipelining to the integrated CPU-GPU circuits, shared memory eliminates most delays, allowing faster task communication and the possibility of computation offloading on the stale CPU cores. Such systems have not significantly increased performance, limited by cache insufficiency. So, the additional investigations at the field of software pipelining concerning GPU.

3. The aim and objectives of the study

The object of the study is the software pipelining technique applied to the GPU.

The purpose of this study is to review the software pipelining approach, its advantages and disadvantages, the techniques that could be used in it, including both instruction-level and decoupled versions, and to find out the effectiveness of this approach for the GPU. To achieve this aim, the following tasks are set:

— Reviewing the software pipelining technique principles and measurements.

— Estimating the existing software pipelining optimisation technique's efficiency.

— Setting up the theoretical background for applying the *C*-slowing technique to increase software pipelining performance.

4. The study materials and methods of investigation

The object of the study is parallel computational processes in the parallel programmable computers. These computers can have superscalar, SIMD or MIMD architectures.

The problem is that the modern GPU usually could not be programmed in assembly language by the user due to the policy of the GPU manufacturers. However, such programming is needed for the highest loading of processing units of this GPU.

The hypothesis is that using new approach to the GPU programming based on the software pipelining using the transformations of Synchronous DataFlow (SDF) can provide more effective programs for GPU designed by the usual programmer in the high level language.

5. Results of *C*-slow retiming investigation

The overviewed software pipelining implementations and techniques, including modulo scheduling, prioritising initiation, and interval minimisation, are considered a primary optimisation approach. However, a lower initiation interval leads to higher register pressure, thus limiting the performance of the pipeline. Moreover, the dense data dependencies between algorithm iterations limit the software pipelining effectiveness. Integrating the *C*-slowing approach into software pipelining is proposed to overcome this adversity. Analysing this technique via the synchronous dataflow model is shown below.

SDF is the graph model for dataflow algorithm representation. SDF is a graph whose nodes represent the operators and edges represent the dataflows. The edge can have either zero delay or a delay, like the FIFO buffer. These algorithms are distinguished because they perform cyclic calculations when the number of processed data in each iteration is stable [30]. Note that SDF or its modifications can represent most of the algorithms implemented in the GPU.

The algorithm represented by the folded SDF performs the same calculations as the original SDF but with a reduced number of nodes [31]. A specific situation of SDF folding occurs when *C* equal SDFs are combined in a single folded SDF. This method is called *C*-slow retiming or *C*-slowing. This method is widely used in hardware design [32]. However, SDF is a multipurpose model. Therefore, *C*-slowing can be effectively implemented in GPU programming.

Consider the following loop nest, which is programmed for a single GPU processing unit (PU), as in Figure 4.

```
for (i = 0; i < N; i++;) {
     r1 = 0;
     for (j = 0; j < M; j++) {
          r1 = c*r1 + a[i,j];
          b[i,j] = r1;
     }
}
```

Fig. 4: Loop example

The program presented in Figure 4 performs a simple filtering algorithm. Here, N filters are executed in parallel. SDF in Figure 5 represents this algorithm.

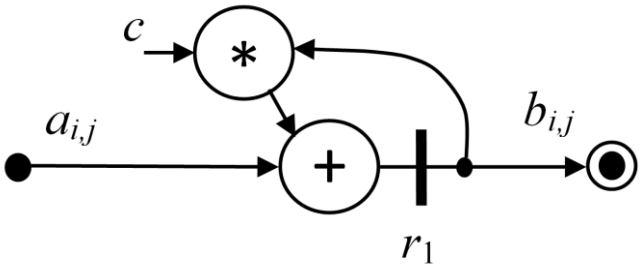

Fig. 5. SDF of the filtering algorithm

The directed edge of SDF represents a dataflow. The bar that loads the edge represents a delay to a single algorithm period. This period in the example is equal to one iteration execution time. So, it represents a register that stores the variable r_1. The bold point and the point in a circle represent the input and output nodes. The data $a_{i,j}$ and $b_{i,j}$ are inputted and outputted through this SDF, respectively. The nodes marked by plus and asterisk mean the addition and multiplication operators, respectively. Such a node outputs its result immediately when the data are present at its inputs.

According to the SDF theory, the critical path of this model goes through the cycle containing the operations nodes and delays [33]. This critical path causes delays in loading and storing the variable $s_i = c \cdot s_i + a_{i,j}$ and delays in addition and multiplication. The iterative loop of the program contains additional delays in loading and storing the data $a_{i,j}$ and $b_{i,j}$ However, the program pipelining and the hardware pipelined memory access in the GPU partially hide the latent data loading and storing delays.

Consider the algorithm loop period T and PU loading ratio Q estimation for this algorithm.

$$T = \max(d_C, d_M + n_{RW} - 1 - \alpha d_C), \quad (1)$$

where d_C is the instruction number in the loop core after compilation, i.e., the approximated number of clock cycles, in which calculations and control load the PU, d_M is the latent delay of the memory access, n_{RW} is the number of accesses to memory in a single iteration, α is the fraction of the number d_C, in which the program pipelining hides the latent delay d_M. Note that the value d_M is the sum of only arithmetic and transfer operation delays because of the loop control, and the array addressing is executed by specific address generators in the GPU. Then, the PU loading ratio is equal to

$$Q = dC / T. \quad (2)$$

In the example above, the variables are $n_{RW} = 2$, $d_C = 5$, $\alpha \approx 0.6$. When data is in local memory, the value is $d_M = 10$ in the GPU PTX architecture. Therefore, $d_M > d_C$. As a result, according to (1) and (2), the iteration period and the PU loading for this example are

$$T_1 = d_M + 1 - 0.6 d_C = 8; \quad Q_1 = d_C / T_1 = 0.625. \quad (3)$$

C-slowing of the algorithm in Figure 5 consists of multiplying C times all delays in the SDF edges. The modified algorithm executes C exemplars of the original algorithm in parallel but with a

slowing in C times. This fact explains the origin of this method's name. Consider the C-slowing when $C = 2$. Then the resulting SDF is shown in Figure 6.

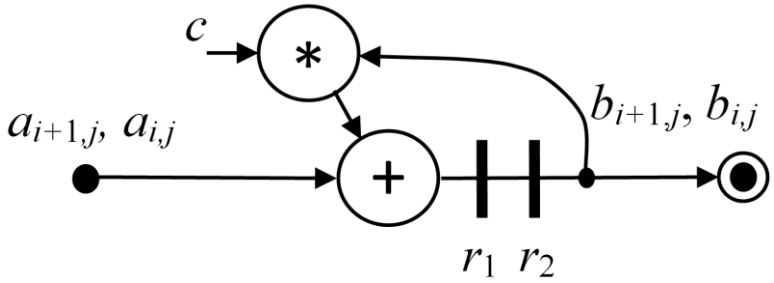

Fig. 6. SDF of the algorithm after C-slowing

Here, the data are loaded and stored sequentially by the couples: $a_{i+1,j}$, $a_{i,j}$ and $b_{i+1,j}$, $b_{i,j}$ in a single iteration. The accumulated result is stored in register $t1$ and its previous value is rewritten in register $r2$ for each iteration. The respective program sketch is as in Figure 7.

```
for  (i = 0; i < N; i+=2) {
     r1 = 0;
     r2 = 0;
     for (j = 0; j < M; j++) {
          r1 = c*r2 + a[i,j];    r2 = r1;
          b[i,j] = r2;
          r1 = c*r2 + a[i+1,j]; r2 = r1;
     b[i+1,j]=r2;
     }
}
```

Fig. 7: C-slowed code

The coefficients are $n_{RW} = 4$, $\alpha \approx 0.6$ and $d_C = 12$ in Figure 7. As a result of C-slowing, according to (1) and (2), the iteration period and the PU loading for this example are $T_2 = 12$ and $Q_2 = 1$. So, the PU loading is increased to its maximum value, and the throughput is increased in $T_1C/\ T_2 = 1.33$ times.

6. Discussion of result

Parallel computations have garnered significant attention, provoking the development of newer and more powerful GPUs and extending their role outside the graphics field. In order to use GPU resources optimally, practical algorithms need to be discovered and analysed. The software pipelining approach has been investigated to analyse the possibilities for improving GPU throughput.

Figure 6 and Figure 7 demonstrate that the C-slowing method is rather effective one, at least, in this example. The calculated improvements are explained below. The memory access time d_M in the GPU is longer than the register access. The example considers that PUs belong to a single warp of GPU, and PUs do not access the common memory cells. Otherwise, the factor d_M increases to tens and hundreds, decreasing the value Q_1 dramatically. Therefore, the program has a short loop core and is inefficient. Moreover, C-slowing increases this core, at least in C times, supporting the computing of groups of C data. This aspect makes the proposed method similar to the modulo-scheduling method described above. However, C-slowing seems to be a more formal method based on the theory of SDF algorithms. However, these aspects need deeper investigation.

7. Conclusion

Reviewing the software pipelining technique principles and measurements shows that the software pipelining on the GPU lacks scientific attention. Further investigations and adjustments are required to determine its capabilities. Future hardware improvements may also provide the necessary changes for the approach to overcome its limitations and show significant performance increases on the GPU.

When estimating the existing software pipelining optimisation technique's efficiency it was instantiated that the C-slowing technique is rather effective to optimise software pipelining. The algorithm's theoretical evaluation has shown a significant performance improvement prospect. However, further practical measurements are required.

The initial theoretical background for applying the C-slowing technique shows the increase of software pipelining performance. However, the additional investigations at this field are needed. For this purpose, the design of the framework for the GPU software pipelining for the artificial intelligence program improvement is planned.

8. Funding

This research was supported in part by the National Research Foundation of Ukraine (NRFU), grant 2025.203/0100.

References

[1] G. Huang et al. "ALCOP: Automatic Load-Compute Pipelining in Deep Learning Compiler for AI-GPUs". In: *Proceedings of Machine Learning and Systems*. Ed. by D. Song, M. Carbin, and T. Chen. Vol. 5. Curan, 2023, pp. 680–694. https://doi.org/10.48550/arXiv.2210.16691.

[2] P. Sakdhnagool, A. Sabne, and R. Eigenmann, "RegDem: Increasing GPU performance via shared memory register spilling," arXiv preprint arXiv:1907.02894, 2019, https://doi.org/10.48550/arXiv.1907.02894.

[3] S. Darabi et al. "Morpheus: Extending the last-level cache capacity in GPU systems using idle GPU core resources". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2022, pp. 228–244, URL: https://doi.org/10.1109/MICRO56248.2022.00029.

[4] V. H. Allan, R. B. Jones, R. M. Lee, S. I. Allan. "Software pipelining." ACM Computing Surveys (CSUR), Vol. 27, No. 3, pp. 367 – 432. https://doi.org/10.1145/212094.212131.

[5] P. Faraboschi, J. Fisher, C. Young. "Instruction scheduling for instruction-level parallel processors". In: Proceedings of the IEEE, V. 89 (Dec. 2001), pp. 1638–1659. https://doi.org/10.1109/5.964443.

[6] K. Ebciŏglu. "A compilation technique for software pipelining of loops with conditional jumps". In: *Proceedings of the 20th annual workshop on Microprogramming*. 1987, pp. 69–79. https://doi.org/10.1145/255305.255317.

[7] Y. Zhang et al. "Clustered Decoupled Software Pipelining on Commodity CMP". In: *Department of Information Science, Graduate School of Engineering*, Utsunomiya University, Japan (2008). https://doi.org/10.1109/ICPADS.2008.113.

[8] E. Raman et al. "Parallel-stage decoupled software pipelining". In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization. CGO '08*. Boston, MA, USA: Association for Computing Machinery, 2008, pp. 114–123. https://doi.org/10.1145/1356058.1356074.

[9] D. C. S. Lucas, G. Araujo. "The Batched DOACROSS loop parallelization algorithm". In*: 2015 International Conference on High Performance Computing & Simulation (HPCS)*. 2015, pp. 476–483. https://doi.org/10.1109/HPCS.2015.47.

[10] A. Zou et al. "RTGPU: Real-Time GPU Scheduling of Hard Deadline Parallel Tasks With Fine-Grain Utilization". In: *IEEE Transactions on Parallel and Distributed Systems* V.34. No5 (2023), pp. 1450–1465. https://doi.org/10.1109/TPDS.2023.3235439.

[11] Y. E. Wang, G-Y. Wei, D. Brooks. "Benchmarking TPU, GPU, and CPU Platforms for Deep Learning". 2019. arXiv: 1907.10701 [cs.LG]. https://doi.org/10.48550/arXiv.1907.10701.

[12] Z. Jia et al. “Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking”. 2018. arXiv: 1804.06826[cs.DC]. https://doi.org/10.48550/arXiv.1804.06826.

[13] C. M. Wittenbrink, E. Kilgariff, A. Prabhu. “Fermi GF100 GPU architecture”. In: *IEEE Micro* 31.2 (2011), pp. 50–59. https://doi.org/10.1109/MM.2011.24.

[14] J. Ghorpade. “GPGPU Processing in CUDA Architecture”. In: *Advanced Computing: An International Journal* 3.1 (Jan. 2012), pp. 105–120. https://doi.org/10.5121/acij.2012.3109.

[15] L. Hu, X. Che, S-Q. Zheng. “A Closer Look at GPGPU”. In: *ACM Comput. Surv.* Vol. 48. No. 4 (Mar. 2016). https://doi.org/10.1145/2873053.

[16] Z. Zheng et al. “VersaPipe: A Versatile Programming Framework for Pipelined Computing on GPU”. In: *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2017, pp. 587–599. https://doi.org/10.1145/3123939.3123978.

[17] C. Oh et al. “GOPipe: A Granularity-Oblivious Programming Framework for Pipelined Stencil Executions on GPU”. In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques. Association for Computing Machinery*, 2020.https://doi.org/10.1145/3410463.3414656.

[18] N. C. Crago et al. “WASP: Exploiting GPU Pipeline Parallelism with Hardware-Accelerated Automatic Warp Specialization”. In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2024, pp. 1–16. https://doi.org/10.1109/HPCA57654.2024.00086.

[19] S. Raskar et al. “Implementation of Dataflow Software Pipelining for Codelet Model”. In: *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering. ICPE ’23.* Coimbra, Portugal: Association for Computing Machinery, 2023, pp. 161–172. https://doi.org/10.1145/3578244.3583734.

[20] H. Wei et al. “Minimizing communication in rate-optimal software pipelining for stream programs”. In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization. CGO ’10.* Toronto, Ontario, Canada: Association for Computing Machinery, 2010, pp. 210–217. https://doi.org/10.1145/1772954.1772984.

[21] S. Laine, T. Karras. “High-performance software rasterization on GPUs”. In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. HPG ’11*. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2011, pp. 79–88. https://doi.org/10.1145/2018323.2018337.

[22] M. Gordon, W. Thies, S. Amarasinghe. “Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs”. In: *ASPLOS XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems.* Oct. 2006, pp. 151–162. https://doi.org/10.1145/1168857.1168877 .

[23] R. G. Singh, C. Scholliers. “Gaiwan: A size-polymorphic typesystem for GPU programs”. In: *Science of Computer Programming* Vol. 230 (2023), p. 102989. https://doi.org/10.1016/j.scico.2023.102989.

[24] A. Udupa, R. Govindarajan, M. J. Thazhuthaveetil. “Software Pipelined Execution of Stream Programs on GPUs”. In: *2009 International Symposium on Code Generation and Optimization*. 2009, pp. 200–209. https://doi.org/10.1109/CGO.2009.20.

[25] J. M. Codina, J. Llosa, A. Gonz´alez. “A comparative study of modulo scheduling techniques”. In: *ICS ’02*. New York, USA: Association for Computing Machinery, 2002, pp. 97–106. https://doi.org/10.1145/514191.514208.

[26] P. Pfahler, G. Piepenbrock. “A comparison of modulo scheduling techniques for software pipelining”. In: *Compiler Construction*. Ed. by Tibor Gyim´othy. Berlin, Heidelberg: Springer, 1996, pp. 18–32. https://doi.org/10.1007/3-540-61053-7_50.

[27] A. Marongiu, V. Nelis, P. Yomsi. “Manycore Platforms”. In: *High Performance Embedded Computing*. 2022, pp. 15–32. https://doi.org/10.1201/9781003338413-2.

[28] J. Hestness, S. W. Keckler, D. A. Wood. “GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors”. In: *2015 IEEE*

International Symposium on Workload Characterization. 2015, pp. 87–97. https://doi.org/10.1109/IISWC.2015.15.

[29] D. Gerzhoy, D. Yeung. “Pipelined CPU-GPU Scheduling to Reduce Main Memory Accesses”. In: *Proceedings of the International Symposium on Memory Systems. MEMSYS ’21.* Washington DC, USA: Association for Computing Machinery, 2023. https://doi.org/10.1145/3488423.3519319.

[30] E. A. Lee and D. G. Messerschmitt. “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing”. In: *IEEE Transactions on Computers*. Vol. C-36. No.1 (Jan. 1987), pp. 24–35. https://doi.org/10.1109/TC.1987.5009446.

[31] K. K. Parhi, C. Y. Wang, and A. P. Brown. “Synthesis of control circuits in folded pipelined DSP architectures”. In: *IEEE Journal of Solid-State Circuits*. Vol. 27. No. 1, Jan. 1992, pp. 29–43. https://doi.org/10.1109/4.109555.

[32] A. Sharma, C. Ebeling, and S. Hauck. “PipeRoute: a pipelining-aware router for reconfigurable architectures”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. Vol. 25. No. 3, Mar. 2006, pp. 518–532. https://doi.org/10.1109/TCAD.2005.853691.

[33] K. K. Parhi. “Algorithm transformation techniques for concurrent processors”. In: *Proceedings of the IEEE*. Vol. 77. No. 12, Dec. 1989, pp. 1879–1895. https://doi.org/10.1109/5.48830.

УДК 004.8 : 004.94

МЕТОД ПРОГРАМНОЇ КОНВЕЙЄРИЗАЦІЇ ДЛЯ ГРАФІЧНИХ СПІВПРОЦЕСОРІВ

Артемій Вінокуров
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна
https://orcid.org/0009-0006-7861-5138

Анатолій Сергієнко
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна
https://orcid.org/0000-0001-5965-1789

Графічні співпроцесори (GPU) відіграють значну роль у високопродуктивних обчисленнях, включаючи штучний інтелект. Однак апаратне забезпечення GPU часто недовантажене. Це змушує збільшувати обсяг апаратного забезпечення GPU для підтримки високої пропускної здатності для виконання завдань. І тому низьке завантаження ресурсів GPU залишається актуальною проблемою, яка вимагає вирішення. Важливо шукати методи, що підвищують завантаження GPU.

Об'єктом дослідження є обчислювальні процеси в сучасних процесорах, особливо в GPU. Метою цього дослідження є огляд підходів удосконалення програмного забезпечення за допомогою програмної конвеєризації, включаючи як версії конвеєризації на рівні команд, так і на рівні програмних потоків, а також оцінка ефективності цього підходу для GPU.

Було розглянуто архітектурні вимоги до застосування конвеєрної обробки програмного забезпечення, розглянуто оригінальне формулювання та історичний розвиток підходу, було досліджено різні рівні паралелізму для реалізації конвеєрної організації програмного забезпечення. Нарешті, було запропоновано метод ресинхронізації з уповільненням як метод оптимізації для подолання недоліків недостатнього використання обчислювальних ресурсів.

Виявлено достатню кількість досліджень, присвячених конвеєрному виконанню програмного забезпечення для реалізацій на GPU. Хоча існуючі роботи розглядають можливості цього методу, вони часто ігноруються на відміну від простіших методів багатопотокової обробки. Однак, в роботах визначили вирішальним обмежувальним фактором обчислювальних ресурсів як обмеження, спричинене перевантаженням пам'яті, зокрема регістрами конвеєрної обробки. Для вирішення цієї проблеми було запропоновано та теоретично оцінено підхід ресинхронізації з уповільненням. Він продемонстрував можливе збільшення завантаження GPU більш ніж на 30% для аналізованого алгоритму, що доводить його перспективи застосування.

На завершення, підхід до конвеєрної обробки програмного забезпечення демонструє непоганий потенціал для оптимізації алгоритмів для GPU, що потребує подальшого дослідження. Ресинхронізація з уповільненням може бути використана для вирішення проблеми прискорення обчислень на GPU.

Ключові слова: програмна конвеєризація, графічний співпроцесор, ресинхронізація, граф синхронних потоків даних.