Solving Large Problem Sizes of Index-Digit Algorithms on GPU: FFT and Tridiagonal System Solvers

Adrián P. Diéguez, Margarita Amor, Jacobo Lobeiras and Ramón Doallo

Abstract—Current GPUs (*Graphics Processing Units*) are capable of obtaining high computational performance in scientific applications. Nevertheless, programmers have to use suitable parallel algorithms for these architectures and usually have to consider optimization techniques in the implementation in order to achieve said performance. There are many efficient proposals for limited-size problems which fit directly in the shared memory of CUDA GPUs, however, there are few GPU proposals that tackle the design of efficient algorithms for large problem sizes that exceed shared memory storage capacity. In this work, we present a tuning strategy that addresses this problem for some parallel prefix algorithms that can be represented according to a set of common permutations of the digits of each of its element indices [1], denoted as Index-Digit (ID) algorithms. Specifically, our strategy has been applied to develop flexible Multi-Stage (MS) algorithms for the FFT (Fast Fourier Transform) algorithm (*MS-ID-FFT*) and a tridiagonal system solver (*MS-ID-TS*) on the GPU. The resulting implementation is compact and outperforms other well-known and commonly used state-of-the-art libraries, with an improvement of up to 1.47x with respect to *NVIDIA's* complex *CUFFT*, and up to 33.2x in comparison with *NVIDIA's CUSPARSE* for real data tridiagonal systems.

Index Terms—GPU, CUDA, Tuning, Tridiagonal systems, FFT, Medium problem sizes, CUSPARSE

1 Introduction

In recent years, GPUs (Graphics Processing Units) have experienced a noticeable increase in their relevance and usage in high performance computing. Nevertheless, programmers have to use suitable parallel algorithms for these architectures that also require special languages, such as NVIDIA CUDA or OpenCL, and ultimately, have to consider optimization techniques in the implementation in order to achieve high performance. Furthermore, the efficient solution on one single GPU of large problem sizes, which exceed limited-size problems that fit directly into the high bandwidth of GPU scratchpad memory (called shared memory in CUDA), is still an important challenge.

The algorithms examined here are described using a parallel prefix approach [2], one of the most popular parallel paradigms. These algorithms are basically regular algorithms whose communication pattern does not depend on execution values; it is given by a linear function, which is well suited to GPU architectures. Furthermore, each resulting element is a combination of previous results of other elements with common calculations that may be reused. Some parallel prefix algorithms may be represented according to a set of common permutations of the digits of each element index [1], denoted as Index-Digit (ID) algorithms. Specifically in this work, Index-Digit algorithms are used for solving the FFT and tridiagonal systems.

• *The authors are with the Computer Architecture Group, University of A Coruña, Spain*
E-mail: adrian.perez.dieguez, margarita.amor, jacobo.lobeiras@udc.es, ramon.doallo@udc.es

Manuscript received ; revised .

The *FFT* is a highly important operation for many applications, such as image and digital signal processing, filtering, compression or partial differential equation resolution. There are a number of auto-tuning proposals for *GPUs*, which achieve high performance, such as [3], [4], [5]. Specifically, approaches focused on large 1D FFT on a single coprocessor include [3], [6], [7]. Another proposal for solving this problem in a sparse format is presented in [8]. However, the most widely used and well-known *GPU* implementation is *NVIDIA's CUFFT* [9].

Tridiagonal systems are types of linear equation systems which are used in many applications, such as fluid simulation or heat conduction and diffusion equations. There are a number of *GPU* tridiagonal solvers implementations based on different algorithms. Many of these can be only applied to problems with many independent small matrices that are stored in the GPU shared memory, such as [10], [11], where parallelism is inherent and there is no partitioning overhead. In [12], the authors first recognized that partitioning is fundamental for solving a single large matrix on GPUs, applying a hybrid PCR [13] - Thomas [14] algorithm. Despite carefully selecting switch points between computation stages, this algorithm suffers from a computation overhead. Argüello et al. [15] proposed a split-and-merge method based on the CR [16] algorithm that reduces the overhead from previous proposals. This split-and-merge approach is later refined in [17]. In [18], the authors present a partitioning method based on the SPIKE [19] algorithm. On the other hand, the diagonal pivoting method for numerical stability is first introduced for GPUs in [20]. Combining QR factorization with Given rotations in [21] improved the previous implementation. In [22], a CR-based approach for solving large-problems is also presented. Finally, *NVIDIA* implements CUSPARSE [23], a library that uses a hybrid CR-PCR implementation with pivoting for solving large-problem sizes. Furthermore, other proposals have recently been developed considering manycore architectures [24] [25], as well as for quasi-tridiagonal systems on GPU [26].

A different approach was presented in [27], which allows the design of ID-algorithms for CUDA-enabled *GPU* architectures with little effort, while obtaining competitive performance with respect to hand-tuned and auto-tuned approaches. This methodology is based on two phases: GPU resource analysis and string operator manipulation. In the first phase, a set of resource factors are obtained from determined premises that will allow high performance to be achieved. In the second phase, CUDA kernels are obtained from a combination of two techniques: index-digit permutations and tuned mapping vector [28]. These techniques adjust the data distribution in the *GPU* according to the resource analysis performed in the first phase. In [27] kernels were built based on BPLG (Butterfly Processing Library for GPUs [29]) functions and focused solely on limited-size problems which fit directly into the shared memory of GPUs.

In this work, we present a strategy that allows the design of efficient large-size ID-algorithms with little effort. This strategy addresses problems whose data cannot be fully stored in the shared memory but which fit into the global memory of a single *GPU*, by partitioning the computation among several stages (multi-stage). It also uses the two-phase methodology presented in [27]. In the first phase, we need to determine the main features which influence GPU performance for these problem sizes and establish a set of premises. Based on these premises, a number of tunable parameters is obtained and, for each algorithm, the optimal values are chosen. In the second phase, CUDA kernels are built and the best performing kernel version is chosen at compile-time with the suitable tunable parameters according to the problem size and target architecture. Specifically, our methodology in this work has been applied to develop flexible algorithms for the Fast Fourier Transform (*MS-ID-FFT*) and a tridiagonal system solver (*MS-ID-TS*) on the GPU. These proposals outperform the CUFFT and CUSPARSE's performance, respectively.

2 Multi-stage Strategy for ID Algorithms

In this section, we present the two-phase methodology proposed on [27] along with the concepts needed to better understand the multi-stage strategy.

Taking CUDA GPU architecture as a reference, thread blocks are distributed by the hardware among the available *Streaming Multiprocessors* (SM) and, depending on the amount of required resources, each SM may be able to simultaneously execute several thread blocks. Each thread block has assigned an amount of shared memory that allows the exchanging of data among threads of the same thread block, whereas threads have associated a certain amount of private registers. There is no explicit CUDA instruction to synchronize thread blocks inside a kernel, as occurs with threads inside a thread block. Nevertheless, when two or more kernels are invoked, there is an implicit global synchronization barrier between each invocation, and each kernel invocation is also known as a *stage*. A more detailed description of *NVIDIA's GPU* architecture can be found in [30].

When several threads need to collaborate in the same task, specifically-designed algorithms are usually required as, in most cases, threads have to communicate their partial results, thus synchronizations are involved. These problems can be classified depending on their size:

- The problem data fit in the shared memory. In this case each problem can be assigned to a single CUDA block, using the shared memory to perform communications. In some cases, even if the problem data exceed the size of the shared memory, it is possible to split data exchange in multiple steps, a technique called shared memory multiplexing [31].
- The problem size is bigger than shared memory but fits into the global memory of a single GPU. In many cases, data are too large to be processed by a single block, and need to be distributed among several blocks. Each thread block locally computes a partial result of the solution. Later, any synchronization mechanism for exchanging partial results within all thread blocks is used and finally, they are combined to obtain the overall solution. However, as has already been mentioned, only threads within the same block can be synchronized in the current CUDA computing model. Data interchange and synchronize mechanishms among blocks are necessary here.
- The problem size is bigger than the global memory of a single GPU. The work is distributed among several streams and GPUs; nevertheless, these problems are beyond the scope of this paper.

The aim of this work is to deal with the second case, problems whose size is bigger than one thread block's shared memory capacity but which still fit into the global memory of a single GPU. Data interchange is performed via global memory, and different options to synchronize thread blocks can be considered:

- *Multi-stage Strategy*. In this case, the work is divided into several kernels; i.e., into several stages. Here, each kernel invocation acts as a global synchronization, as explained above. Thread blocks from each corresponding kernel write their partial results into global memory. As the synchronization mechanism for this data interchange, another kernel is launched with its corresponding thread blocks. These new thread blocks build new partial results using the previous kernel data from global memory. This strategy significantly increases the global memory bandwidth requirements.
- *Dynamic parallelism*. Using dynamic parallelism, a kernel directly from GPU can spawn other kernels. Its main objective is to reduce the overhead of starting and synchronizing kernels. Even considering the

Parameter	*Definition*
$N = r^n$	Problem size.
r^{batch}	Number of problems being simultaneously solved.
$P = r^p$	Number of elements stored in registers per thread.
$B = r^b$	Number of thread blocks per stage, where $B = B_x \cdot B_y$
$L = r^l$	Number of threads per thread block, where $L = L_x \cdot L_y$
$S = r^s$	Number of shared-memory elements per thread block, where $s = n - b_x$ and $s = p + l$
m	Number of computing stages, where $m = \lceil \frac{n}{s} \rceil$

TABLE 1: Description of tuning parameters.

added flexibility, the generated code tends to run slower due to the relocatable device code generation and the use of local memory, global memory accessible only by the thread that declares it, as a call stack [32]. This approach is suitable for problems that require mesh refinement (such as finite element methods) using a dynamic work distribution.

- *Persistent threads*. This is a descentralized sleeping strategy. Each thread block sets a flag when it reaches the intra-block barrier, executing an infinite loop until a master thread block changes the flag value. When all flags are set, the master block resets them and all thread blocks continue execution. This allows thread blocks to be synchronized in a single kernel. A kernel uses, at most, as many thread blocks as can be concurrently scheduled on the SM. Thus, this strategy synchronizes global memory using a single kernel and a constant number of thread blocks. In many cases, the use of persistent threads on GPUs results in performance losses [33]; nonetheless, it has been successfully applied in some optimized libraries, such as CUB [34], as it presents low memory contention.

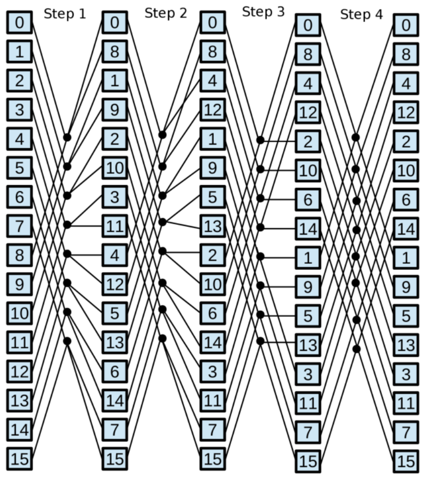
In this work we have used the multi-stage strategy as the synchronization mechanism among thread blocks. This is the same technique used in other libraries, such as CUFFT [9] or the proposal presented in [11]. Despite the increased global memory requirements, if the data exchanges are properly optimized and the workload is properly balanced among the GPU resources, the multi-stage strategy is quite efficient.

In the following subsections, a number of basic concepts introduced by our multi-stage strategy are presented. As we will explain below, ID-algorithms are formulated with a set of *string operators*, which allows us to describe both the behavior of the algorithm and the mapping of data over GPU resources. The objective is to design efficient ID-algorithms that can be easily adapted to the target architecture. With this in mind, string operators are used to efficiently solve large problem sizes.

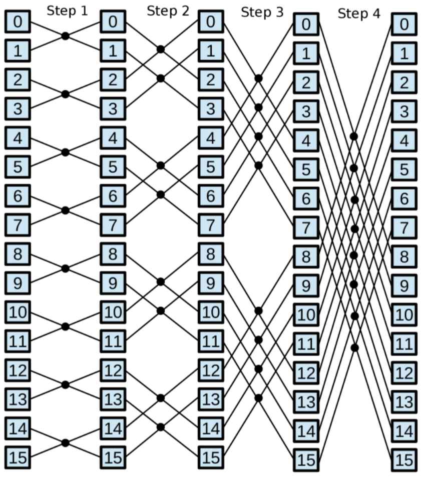
2.1 GPU Resources Utilization Analysis Phase

A description of Index-Digit algorithms is presented in this subsection. All parameters defined for our multi-stage strategy are collected in Table 1.

An Index-Digit algorithm is a kind of regular algorithm for a problem with size $N = r^n$, where r is called *radix*. The



(a)



(b)

Fig. 1: Patterns of Index-Digit algorithms for radix-2 and N=16 (a) Stockham pattern (b) Cooley-Tukey pattern without bit-reversal step

data interchange can be modeled by the rearrangement of a data array according to a common permutation of the digits of each element's index. To this end, data item $x(t)$ with index $t = t_n \cdot r^{n-1} + \cdots + t_2 \cdot r + t_1$ is written as $[t_n \cdots t_2 t_1]$. For example, element $x(5)$ of an arbitrary radix-2 data sequence of $N = 16 = 2^4$ elements, is represented as $[0101]$. Each Index-Digit algorithm may be represented by a directed acyclic graph, called a prefix circuit [35]. An ID-algorithm is processed over several steps where each step has a given number of node computations, which execute the core operation over a number of elements. Although there is computational independence within each step, there are data dependencies among the different steps. The node operator is represented by black circles in the prefix circuit in Figure 1, and the radix r also represents the number of elements that take part in each node computation, computing them in one single step, where N/r is the number of node computations in a single step. But additionally, the size of the problem, N, can be expressed as a power of r, $N = r^n$.

For example, Figure 1 shows two different patterns over $N = 16$ elements. As each node computes 2 elements in a single step, thus $r = 2$ and $N = r^n = 2^4$, it takes 4 steps, using 8 nodes in each step. On the other hand, if $r = 4$ and each node computes 4 elements in a single step, then it would take 2 steps and 4 nodes would be needed in each step. Figure 1 (a) displays the Stockham pattern, used in FFT which distributes the bit reversal operation among the different radix steps. Figure 1 (b) shows the Cooley-Tukey pattern, without the first step, which performs the bit reversal operation.

In [27], kernel data are divided among $B = r^b$ thread blocks, and each of these blocks executes $L = r^l$ threads. A thread performs the calculation of $P = r^p$ data stored in private registers and threads within a thread block have access to $S = r^s$ data stored in shared memory. Henceforth, for the sake of clarity, we will consider the index-digit representation with $r = 2$. Furthermore, r^{batch} number of problems (data sequences) are simultaneously processed in a batch mode by a single invocation to our library. Thus, the mapping of r^{batch} data sequences of size $N = r^n$ is identified with a 5-tuple of the form (n, p, s, l, b).

In our multi-stage proposal of ID-algorithms on GPU, each problem is computed by dividing it into a set of m stages, where each stage executes several steps. Each stage executes a kernel which assigns a part of its corresponding problem to different thread blocks. Here, b is formed by two coordinates $b = (b_x, b_y)$, where r^{b_x} represents the number of used thread blocks per each problem while r^{b_y} represents the number of problems being simultaneously executed on that kernel in a batch mode, also known as segmented execution. Thus, $B_x \cdot B_y$ blocks process the whole batch. Furthermore, each thread performs the computations associated to the node operator. Data are stored in private registers in order to achieve high performance, as register files have lower access latency and higher bandwidth than shared memory. Finally, threads from the same thread block exchange data before the next computing step through shared memory. Specifically, our multi-stage tuning proposal is based on only three parameters (n, p, b_x) given that $s = n - b_x$, as all the data stored in registers also have a copy in shared memory to perform the intra-block memory exchanges; and $b_y = batch$ is given by the batch size, which is only known at runtime. In our proposal, l consists of three coordinates (l_x, l_y, l_z) where the second and third coordinates, (l_y, l_z), are optional. The l parameter can be related with s and p using $s = p + l$.

2.1.1 Premises for Performance Maximization

In this work, large problems are computed over several kernels. When computing several kernels, new parameters influence performance. For example, the number of invoked kernels, the number of steps processed by kernel and the number of elements processed by each kernel. Considering these factors and attempting to improve performance, we define the following premises:

1) *The minimization of the number of stages, m.* Global memory data exchanges are slower than using other memories, such as shared memory, despite implicitly utilizing the L1 and L2 caches. In the multi-stage strategy, data interchange via global memory is the only method for sharing information among kernels, as $n > s$. In addition to this latency, each kernel invocation implies an overhead, even for empty kernels. Thus, the number of stages (kernels) in the multi-stage strategy needs to be minimized. In our proposal, the number of stages is given by the following expression:

$$m = \left\lceil \frac{n}{s} \right\rceil \quad (1)$$

In order to minimize this expression, s must be as large as possible. Each kernel invocation executes as many problem steps as the shared memory allows. Thus, each kernel processes several chunks of S elements (one chunk per thread block). Subsequent kernels will merge elements among chunks until the final result is obtained.

2) *Balancing warp and block parallelism*. In the GPU, the level of parallelism can be supported in terms of the number of thread blocks per *SM* (*SM* block parallelism), or the number of warps per *SM* (*SM* warp parallelism):

   a) *The maximization of block parallelism in each stage* in order to keep processing the maximum amount of simultaneous thread blocks per *SM* (16 in the case of *Kepler* and 32 in the case of *Maxwell*-based *GPUs*). In fact, the *GPU* hardware is able to provide highly satisfactory performance even at lower occupancies (low *SM* warp parallelism) [36], [37].

   b) *The maximization of warp parallelism in each stage*. This premise is focused on increasing the number of warps per *SM*.

   Our proposal attempts to strike a balance between the maximization of warp and block parallelism. In order to increase this parallelism, we need to limit the factors that reduce the SM parallelism, such as the number of registers used by each thread or the amount of shared memory required by thread block.

3) *Increasing the computational load per thread*. Both r and P parameters are closely related. Note that r is a feature of the algorithm which represents the number of data computed in each node. However, if the target architecture allows more than r elements to be stored in registers, without SM occupancy penalization, it may be interesting to process more node computations per thread, without modifying the base r of N. In this case, each thread processes P elements in $\frac{P}{r}$ radix-r nodes. Increasing either P or r means processing more elements per thread. It influences the number of steps taken and the number of threads which process a problem, and reduces the number of synchronization barriers. Thus, larger values obtain higher performance. Nevertheless, their increase may also require too many registers per thread, resulting in local memory spilling and the minimization of parallelism. In this

work, $\frac{P}{r}$ radix-r nodes are easily integrated in a single radix-P node, reducing the number of steps taken.

Combining all of these premises is not easy. Firstly, s needs to be increased in order for there to be fewer stages (Premise 1). This is fundamental since it avoids launching several kernels, synchronizations and reads/writes to global memory. This increment entails more elements being processed by a single kernel. However, the increase of s may decrease the SM block parallelism (Premise 2.a). Each SM has a fixed amount of shared memory partitioned among the thread blocks, thus the amount of shared memory required by each thread block limits the SM block parallelism. In order to achieve Premise 2.b, keeping the number of thread blocks constant, l must be raised. Due to the equation $s = p + l$, there are two options: either decreasing p and keeping s constant or increasing s and keeping p constant. In the former option, block parallelism remains the same, reducing the workload per thread (which implies more steps, loop iterations and shared memory accesses), whereas the opposite happens in the latter. In addition to this, each SM also has a register file partitioned among threads. Decreasing register consumption implies better warp occupancy. However, p should be high if Premise 3 is to be achieved. The maximum number of concurrent warps and thread blocks per SM depends on the architecture. In the case of *Kepler*, the total amount of registers per *SM* is 65536 and the amount of shared memory per *SM* is 48 KB, enabling up to 16 concurrent thread blocks and 64 concurrent warps. The number of registers used by each thread is assigned at compile time. In hardware with *CUDA* capabilities 2.x or 3.0, it is not possible to assign more than 63 registers to the same thread. Hardware with *CUDA* capabilities 3.5 supports up to 255 registers per thread. If the kernel requires more registers than those supported by the architecture, local memory spilling will be generated. This means using global memory for placing values instead of registers, paying the penalty of global memory latency. Regarding *Maxwell GPUs*, the architecture has 96 KB of shared memory per SM and can use up to 48 KB per thread block, with the register file size remaining constant. It enables up to 32 concurrent thread blocks and 64 concurrent warps.

Our main objective in the optimization of these algorithms is to find a trade-off between premises for each problem on each architecture in order to achieve the highest possible performance.

In the case of an arbitrary N, our methodology is easily extended. If N is not a power of r, then it can be expressed as $r^n + N'$. On the one hand, r^n is computed following the explained methodology. On the other hand, the smallest power of r able to compute N' is executed, and then r^n and N' data are integrated into one step.

2.2 CUDA Kernel Optimization Phase: String Operators and Mapping Vector

This section describes the use of mapping vectors based on the Index-Digit representation [28]. The mapping vector is a compact representation of the data distribution on the system memory hierarchy. A mapping vector divides the Index-Digit representation into different fields which are used to assign resources of the CUDA GPU (e.g. as thread block, thread or registers) to the specific data item to be treated by the GPU. At the beginning and the end of the algorithm, data reside in global memory; however, data are moved among different resources in the GPU during the execution. The string operator provides a precise description of the data reordering, being useful in the design and optimization of different algorithms. Furthermore, the string operator makes it possible to obtain an index-digit representation in each step of the algorithm. Further information about string operator properties can be found in [38].

Data sequences are stored in the *GPU*'s global memory with a consecutive data distribution according to the following mapping vector:

$$[\ t_{n+batch} \cdots t_{n+1}\ t_n \cdots t_1\ ] \tag{2}$$

This means that data with size N will be stored consecutively in global memory. Hence, the first data sequence of the batch will start at location 0, the second at location N, and the i-th problem of the batch at location $i \times N$.

The mapping vector for data on the *SM* resources that we consider is

$$[\ \underbrace{t_{n+batch} \cdots t_{s+1}}_{b}\ \overbrace{\underbrace{t_s \cdots t_{l+1}}_{p}\ \underbrace{t_l \cdots t_1}_{l}}^{s}\ ] \tag{3}$$

Firstly, this means that each thread block $i = [t_{n+batch} \cdots t_{s+1}]$ processes S items of data which are stored in shared memory, and secondly, thread $j = [t_l \cdots t_1]$ within a thread block processes P items of data where datum $[t_n \cdots t_{l+1}\ t_l \cdots t_1]$ is stored on the register $[t_s \cdots t_{l+1}]$ of thread j. Note that consecutive data belong to different threads.

However, the data distribution of a problem could change depending on the implementation design for a given target architecture. For instance, the previous example can be also expressed with the following mapping vector:

$$[\ \underbrace{t_{n+batch} \cdots t_{s+1}}_{b}\ \overbrace{\underbrace{t_s \cdots t_{p+1}}_{l}\ \underbrace{t_p \cdots t_1}_{p}}^{s}\ ] \tag{4}$$

In this case, each thread $j = [t_s \cdots t_{p+1}]$ within a thread block, processes P consecutive data stored in registers.

Figure 2 depicts an example of mapping the data to the GPU resources when $s = 9$, $p = 4$ and $b_x = 2$ for the case $r = 2$ and $n = 11$. Each thread block receives a set of elements, $2048/4 = 512$, which are stored in shared memory, and evenly distributed to the registers among 32 threads ($l = 9 - 4$). The mapping vector for this example would be:

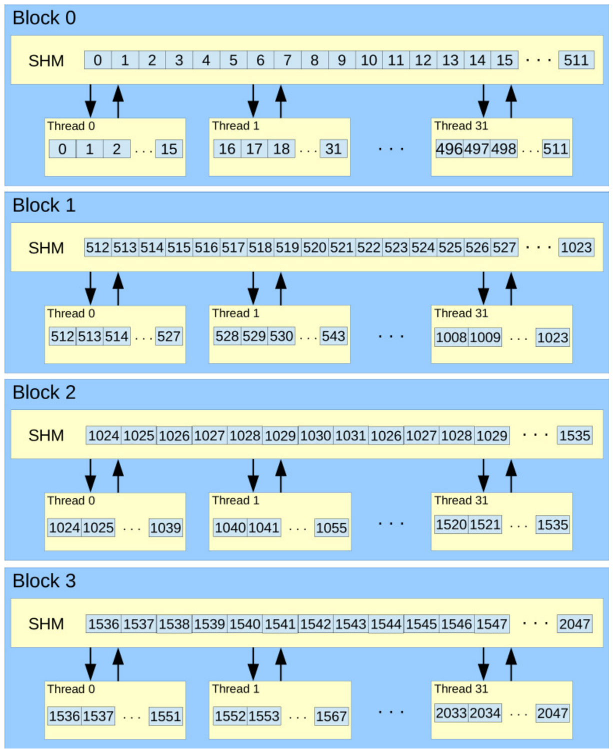

Block 0
SHM 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ··· 511
Thread 0 0 1 2 ... 15
Thread 1 16 17 18 ... 31
Thread 31 496 497 498 ... 511
Block 1
SHM 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 ··· 1023
Thread 0 512 513 514 ... 527
Thread 1 528 529 530 ... 543
Thread 31 1008 1009 ··· 1023
Block 2
SHM 1024 1025 1026 1027 1028 1029 1030 1031 1026 1027 1028 1029 ··· 1535
Thread 0 1024 1025 ··· 1039
Thread 1 1040 1041 ··· 1055
Thread 31 1520 1521 ··· 1535
Block 3
SHM 1536 1537 1538 1539 1540 1541 1542 1543 1544 1545 1546 1547 ··· 2047
Thread 0 1536 1537 ··· 1551
Thread 1 1552 1553 ··· 1567
Thread 31 2033 2034 ··· 2047

Fig. 2: Data mapping with $r = 2$, $n = 11$, $s = 9$, $p = 4$ and $b_x = 2$.

Operator	*Definition*
Node	Υ_i^r, with $1 \le i \le n$, computes r data elements whose index differs in the i-th digit.
Perfect Unshuffle	$\Gamma_{i,j}[t_n \cdots t_1] = [t_n \cdots t_{i+1} \underline{t_j t_i \cdots t_{j+1}} t_{j-1} \cdots t_1]$
General Unshuffle	$\Gamma_{i,j,k,l}[t_n \cdots t_1] = [t_n \cdots t_{i+1} \underline{t_l t_i \cdots t_{j+1}} t_{j-1} \cdots t_{k+1} \underline{t_j t_k \cdots t_{l+1}} t_{l-1} \cdots t_1].$
Digit Reversal	$\rho_{i,j}[t_n \cdots t_1] = [t_n \cdots t_{i+1} \underline{t_j t_{j+1} \cdots t_i} t_{j-1} \cdots t_1].$

TABLE 2: Description of string operators.

$$[\, \underbrace{t_{batch+11} \cdots t_{12}}_{b_y} \; \underbrace{t_{11} t_{10}}_{b_x=2} \overbrace{\underbrace{t_9 \cdots t_5}_{l=5} \; \underbrace{t_4 \cdots t_1}_{p=4}}^{s=9} \,] \tag{5}$$

For example, element $1041 = [\cdots \underbrace{10}_{b_x} \underbrace{00001}_{l} \underbrace{0001}_{p}]$ is processed by thread 1 ($l = 00001$) in block 2 ($b_x = 10$) and stored in register 1 ($p = 0001$) of that thread.

We define two types of operators which correspond to computations and data permutations, respectively. All of these are formally defined in Table 2, wherein the modified digits are underlined. To write the expressions of the string operators we follow the convention of composing operators from left to right. For example, in the string operator $\phi_1\phi_2$, we first execute ϕ_1 and then, ϕ_2. First, we define the operator that represents the computations.

Definition 1. The node operator, Υ_i^r, with $1 \le i \le n$ where $n = log_r N$, reads those sets of r data items whose position differs precisely in their i-th digit, performs an operation over them and writes r results.

Depending on the operation, each node function will be defined with its own behavior for each algorithm. This specialization only affects the implementation details, but not the methodology design. In general, to simplify the notation when using the basic radix-2 algorithm, the expression of this operator will be referred to simply as Υ_i instead of Υ_i^2. Furthermore, in order to clarify the explanation, we keep the index-digit representation with $r = 2$.

The second type of operators represents data permutations.

Definition 2. The perfect unshuffle operator $\Gamma_{i,j}$, $i \ge j$, performs a cyclic shift to the right between the i-th and j-th digits of the index-digit representation of the data.

We also define a generalization of this operator, $\Gamma_{i,j}^m$. Instead of performing a single cyclic shift to the right, it will perform m consecutive shift operations, such as $\Gamma_{i,j}^2 = \Gamma_{i,j}\Gamma_{i,j}$. For instance, $\Gamma_{7,2}^2[\, t_8\; t_7\; t_6\; t_5\; t_4\; t_3\; t_2\; t_1 \,] = [\, t_8\; \underline{t_3\; t_2\; t_7\; t_6\; t_5\; t_4}\; t_1 \,]$.

Definition 3. The general unshuffle operator $\Gamma_{i,j,k,l}$, $i \ge j \ge k \ge l$, is similar to the previous definition, however it is applied to two digit subfields $\{i \dots j\}$ and $\{k \dots l\}$ of the index-digit representation.

Therefore, data in the range $\{t_{j-1} \cdots t_{k+1}\}$ remain unmodified. For instance, $\Gamma_{8,6,2,1}[\; t_8\;\; t_7\;\; t_6\;\; t_5\;\; t_4\;\; t_3\;\; t_2\;\; t_1 \;] = [\; \underline{t_1\; t_8\; t_7}\; t_5\; t_4\; t_3\; \underline{t_6\; t_2} \;]$.

Definition 4. The digit reversal operator $\rho_{i,j}$, $i \ge j$, performs the reversal of the digits between the i and j-th digit of the index-digit representation of the data.

For instance, the digit reversal of the sequence $\rho_{7,2}[\; t_8\; t_7\; t_6\; t_5\; t_4\;\; t_3\; t_2\; t_1 \;] = [\; t_8\; \underline{t_2\; t_3\; t_4\; t_5\; t_6\; t_7}\; t_1 \;]$. This operator coincides with its inverse.

Once the algorithm expression is generated with the operators, obtaining the code is quite straightforward. Permutation operators are easily implemented using different strides and offsets when transferring data from different memory spaces. Computation operators are implemented directly from their definition. The implementation makes extensive use of template functions (skeletons) to create several optimized versions, depending on the problem size and the target architecture. Different tables are built, where each problem size represents an entry indicating both how to split the problem over the number of kernels and the optimized performance parameters for each kernel. The library chooses the entry depending on the problem size and target architecture, and kernels are then built with these parameters at compile time, via template metaprogramming. Hence, the user does not have to generate it. Most of the function calls, register loops and redundant move operations will be fully optimized at compile time. Thus, this approach provides generality and usability, generating well performing kernels with little effort, as can be seen in the performance evaluation section (see Section 5).

3 Multistage Index-Digit FFT Algorithm

Our *MS-ID-FFT* proposal is based on the *Stockham* algorithm [39]. Figure 1 (a) displays the pattern of this algorithm with $N = 16$ and radix 2, which provides an output sequence that is digit reversed with respect to the input sequence (a specific bit reversal stage is not required). The *Stockham* pattern can be represented using the following expression:

$$\prod_{i=1}^{n} \Gamma_{n,n-i+1} \Upsilon^{r}_{n}. \tag{6}$$

Before explaining the mapping vector obtained from the previous expression, our *MS-ID-FFT* algorithm is introduced:

1) Each thread loads a small portion of data from global memory into registers using coalescent memory access.
2) Compute the first step using radix-P or a mixed-radix.
3) Exchange data through shared memory.
4) Compute the following step.
5) If no step assigned to this stage remains then stop and data are written to global memory using a permutation; this will contribute to the progressive bit-reverse of the signal required by the FFT algorithm.
6) There are two possibilities:
    a) If all the required stages have been computed the algorithm terminates.
    b) Otherwise, the next kernel re-reads the previous data from the global memory, albeit with a different permutation to ensure that once again it is possible to perform several radix stages. Step 1 is repeated, launching as many kernels as required until the problem has been processed.

3.1 MS-ID-FFT Mapping Vector

Taking the foregoing into account, the mapping vector for distributing data to the GPU resources that we propose is:

$$[\, \underbrace{t_{n+batch} \cdots t_{n+1}}_{b_y} \underbrace{t_n \cdots t_{l+b_x+1}}_{p}\ \underbrace{t_{l+b_x} \cdots t_{l_x+l_y+b_x+1}}_{l_z} \tag{7}$$
$$\underbrace{t_{l_x+l_y+b_x} \cdots t_{l_x+b_x+1}}_{l_y}\ \underbrace{t_{l_x+b_x+1} \cdots t_{l_x+1}}_{b_x} \underbrace{t_{l_x} \cdots t_1}_{l_x} \,]$$

It should be noted that l_x is mapped to the lower part of the index-digits to ensure global memory coalescence (consecutive threads access adjacent memory locations) as mentioned in Point 1 of the proposal details. Each 2^{l_x} of data form a *batch*, which will be processed in parallel as such. The remaining threads, $2^{(l_y+l_z)}$, load the actual data which are processed and transformed in the current stage (Point 2 and 4). In this case there is more flexibility in the memory access pattern as memory coalescence is now guaranteed. However, they will usually control the higher bits of the data sequence in the mapping vector representation. During data exchanges, all thread block data reside temporally in shared memory (Point 3):

$$s = l_x + l_y + l_z + p \tag{8}$$

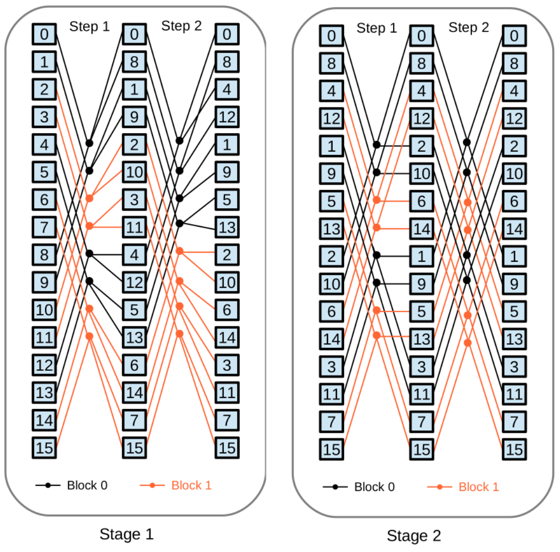


Fig. 3: Distribution of the Stockham pattern with multi-stage proposal for $N = 16$ using the parameters $p = l_y = l_x = 1$ and $b_x = 1$.

Nonetheless, there are some strategies, such as shared-memory multiplexing, which make it possible to reduce the shared memory usage. For instance, in the case of complex data, it is possible to exchange first the real part and then the imaginary part. Another solution would be to perform the data exchange in several stages and only one subgroup of threads can communicate each time. This approach, however, greatly increases the synchronization costs. Furthermore, dynamic indexing refers to an index which the compiler cannot resolve as constant, placing it into local memory instead of registers, with the consequent performance loss. Assigning the value p to the non-zero dimensions that compose l produces simpler indexing expressions that can be easily resolved as constant by the compiler, avoiding poor-performance behaviors such as dynamic indexing.

For example, Figure 3 displays the Stockham pattern of the Figure 1 (a) for $N = 16$ using the parameters $p = l_y = l_x = 1$ and $b_x = 1$. This means that each data sequence is processed in two stages ($m = \lceil \frac{4}{3} \rceil$) by two thread blocks and each block consists of 4 threads ($L = 2^{l_y+l_x} = 2^{1+1}$) which process one node function ($p = 1$). Figure 4 also displays the Stockham pattern but indicating the mapping to the GPU resources and the operation performed in each step. Each thread block loads the corresponding data from global memory, placing them into registers. Then, threads perform the node operation in registers, using shared memory for exchanging elements (stage 1). Once S elements have been processed, data are written to global memory in order to be used for the next kernel. The second kernel repeats the process, writing the final result into shared memory (stage 2). The mapping vector for the example would be:

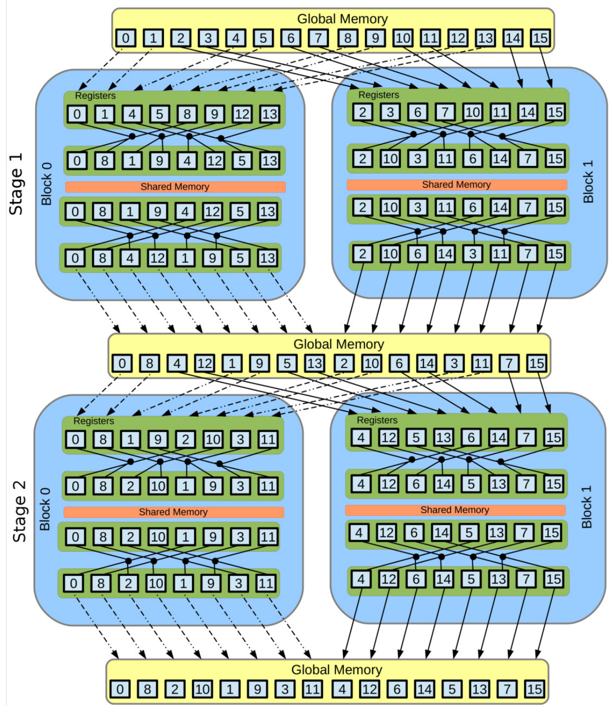


Fig. 4: Mapping of the Stockham pattern on the GPU resources for $N = 16$ using the parameters $p = l_y = l_x = 1$ and $b_x = 1$.

$$[\underbrace{\cdots t_5}_{b_y} \underbrace{\overbrace{t_4}^{p}\overbrace{t_3}^{l_y}\overbrace{t_2}^{b_x}\overbrace{t_1}^{l_x}}_{n}] \tag{9}$$

Regarding the premises explained previously, a trade-off needs to be found between number of stages, SM parallelism and workload per thread. In this work, we consider problem sizes with $12 \leq n \leq 24$. Considering $m = \lceil \frac{n}{s} \rceil$ and Premise 1, s should be as high as possible. On the other hand, taking Premise 2 into account, block parallelism should be maximized. Combining both, the s value that minimizes m when $12 \leq n \leq 24$, and which least affects block parallelism, is $s = 12$. However, FFT node operates with two complex data items (8 bytes for single precision), thus $s = 12$ would involve 32 KB of shared memory, and therefore, each SM could execute a single thread block. In order to avoid this, a shared memory multiplexing technique [31] is applied, firstly exchanging the real part and then exchanging the imaginary part, reducing the shared memory consumption to 16 KB per thread block (3 active blocks per SM); i.e., each element is expressed as a 4-byte datum in shared memory. Taking the remaining parameters into account, several options are available, considering $s = p + l$ and $l_z = l_y = l_x = p$ with $s = 12$. As using $p = 6$ would imply having $l_x = 6, l_y = l_z = 0$, and therefore 2 warps per SM at most, achieving low warp parallelism (Premise 2.b), the value of p has been limited to 4, thus increasing l due to $s = p+l$. As explained, the non-zero dimensions that compose l should coincide with the value p in order to avoid dynamic indexing. There are many options, specifically all combinations that match the $12 = p+l$ equation with $p \leq 4$. Attending to Premises 2 and 3 , the options are, on the one hand, to use $p = 4$ and $l_x = l_y = 4$; or, on the other, to use $p = 3$ and $l_x = l_y = l_z = 3$. Both options are described below as *MS-ID-FFT*.V1 and *MS-ID-FFT*.V2, respectively:

- *MS-ID-FFT*.V1. The mapping vector for the first option, $p = l_x = l_y = 4$, is:

$$\cdots \underbrace{t_n \cdots t_{l+b_x+1}}_{p} \underbrace{t_{l+b_x} \cdots t_{l_x+b_x+1}}_{l_y} \underbrace{t_{l_x+b_x+1} \cdots t_{l_x+1}}_{b_x} \underbrace{t_{l_x} \cdots t_1}_{l_x} \tag{10}$$

The string operator used in this case would be as follows:

$$\left(\rho_{n,n-p+1}\ \Upsilon^p_{n-p+1}\right) \Gamma^p_{n,n-p+1,l+b_x,l_x+b_x+1} \tag{11}$$
$$\left(\rho_{n,n-p+1}\ \Upsilon^p_{n-p+1}\right) \Gamma^p_{n,n-p+1,l_x,1} \left(\rho_{n,n-p+1}\ \Upsilon^p_{n-p+1}\right)$$

In this version, the resource factors used are (p, s, l) = (4, 12, $(4, 4)$). For example, for $n = 13$ the data mapping in the GPU resources is as follows:

$$[\underbrace{\cdots t_{14}}_{b_y} \overbrace{\underbrace{t_{13}t_{12}t_{11}t_{10}}_{p} \underbrace{t_9t_8t_7t_6}_{l_y} \underbrace{t_5}_{b_x} \underbrace{t_4t_3t_2t_1}_{l_x}}^{n}] \tag{12}$$

With the following operator string:

$$(\rho_{13,10}\ \Upsilon^4_{10})\, \Gamma^4_{13,10,9,6}\, (\rho_{13,10}\ \Upsilon^4_{10}) \tag{13}$$
$$\Gamma^4_{13,10,4,1}\, (\rho_{13,10}\ \Upsilon^4_{10})$$

The level of parallelism achieved depends on the GPU architecture on which the proposal is executed. In order to ascertain how many thread blocks and warps are executed, we need to pay attention to the amount of common resources consumed. This version uses $l = 8$; i.e., 256 threads per thread block. In addition to this, it consumes 32 registers per thread ($p = 4$) for storing elements, but taking into account auxiliary variables, the real consumption is between 46 and 54. Furthermore, the shared memory per thread block is between 16.4 KB and 17.4 KB. In the case of Kepler, which can execute up to 16 blocks and 64 warps per SM, this version executes 2 thread blocks and 16 warps in each SM. Regarding Maxwell, which can hold up to 32 active thread blocks and 64 active warps per SM, it achieves 3 blocks and 24 warps .

- *MS-ID-FFT*.V2. The mapping vector for the second option, $p = l_x = l_y = l_z = 3$, is:

$$\cdots \underbrace{t_n \cdots t_{l+b_x+1}}_{p} \underbrace{t_{l+b_x} \cdots t_{l_x+b_x+1}}_{l_z} \underbrace{t_{l_x+l_y+b_x} \cdots t_{l_x+b_x+1}}_{l_y} \underbrace{t_{l_x+b_x+1} \cdots t_{l_x+1}}_{b_x} \underbrace{t_{l_x} \cdots t_1}_{l_x} \tag{14}$$

The string operator used in these case would be as follows:

$$\left(\rho_{n,n-p+1}\ \Upsilon^{p}_{n-p+1}\right)\Gamma^{p}_{n,n-p+1,l+b_x,l_x+b_x+1}$$
$$\left(\rho_{n,n-p+1}\ \Upsilon^{p}_{n-p+1}\right)\Gamma^{p}_{n,n-p+1,l_x,1}$$
$$\left(\rho_{n,n-p+1}\ \Upsilon^{p}_{n-p+1}\right)$$

In this version, the resource factors used are (p, s, l) = (3, 12, $(3,3,3)$). For example, for $n = 13$ the data mapping in the GPU resources is as follows:

$$[\underbrace{\cdots t_{14}}_{b_y}\ \overbrace{\underbrace{t_{13}t_{12}t_{11}}_{p}\ \underbrace{t_{10}t_9t_8}_{l_z}\ \underbrace{t_7t_6t_5}_{l_y}\ \underbrace{t_4}_{b_x}\ \underbrace{t_3t_2t_1}_{l_x}}^{n}] \tag{15}$$

In this case, each problem is executed using 512 threads per thread block, the real number of registers per thread is between 35 and 41, when $p = 3$, and the shared memory per thread block is 16.6 KB. On Kepler, each SM can execute up to 2 blocks and 32 warps; whereas on Maxwell, it executes 3 blocks and 48 warps.

With the following string operator:

$$\begin{array}{l}\left(\rho_{13,10}\ \Upsilon^{4}_{10}\right)\Gamma^{4}_{13,10,9,6}\\ \left(\rho_{13,10}\ \Upsilon^{4}_{10}\right)\Gamma^{4}_{13,10,4,1}\left(\rho_{13,10}\ \Upsilon^{4}_{10}\right)\end{array} \tag{16}$$

As can be seen in both options, the solution is quite regular and easy to understand.

4 Multistage Index-Digit Tridiagonal System Solver Algorithm (MS-ID-TS)

A tridiagonal system is composed of N equations, where each equation E_i, with $i = 1,\cdots,N$, takes the form: $a_ix_{i-1}+b_ix_i+c_ix_{i+1} = d_i$. If $|b_i| \geq |a_i|+|c_i|, \forall i = 1,\cdots,N$, hence the system is known as diagonally dominant. This kind of matrix guarantees numerical stability in most of the algorithms proposed in the literature. Most recent GPU implementations for solving large-problem sizes are based on the CR algorithm in order to avoid computational overheads suffered by other algorithms. However, the CR algorithm still needs $2n-1$ computational steps.

Our *MS-ID-TS* proposal is based on the Wang and Mou algorithm [40], which is based on the same Divide-and-Conquer strategy [41] as the SPIKE algorithm. However, in contrast to the SPIKE algorithm, the diagonalization of each block is performed using the Gaussian elimination method, also reordering the equations in a different way. The Wang and Mou algorithm is a good match for the GPU architecture, offering excellent performance. The computation is divided into n steps, and it follows a pattern similar to the *Cooley-Tukey*, but excluding the initial bit-reversal stage. Figure 1(b) shows the pattern of this algorithm with $N = 16$ and radix-2. This pattern can be represented using the following expression:

$$\prod_{i=1}^{n}\Upsilon^{r}_{i}. \tag{17}$$

Each node operator operates on triads of equations, labeled *L*eft, *C*enter and *R*ight, represented as:

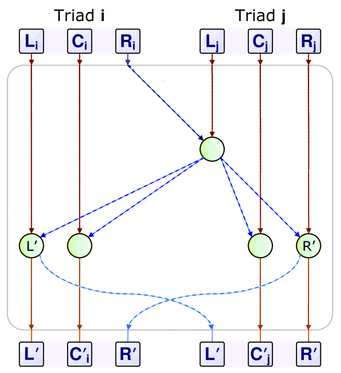


Fig. 5: Node operator in using Wang and Mou algorithm.

$$[i]^{t-1} = [\ \underbrace{E^{t-1}_{q\cdot 2^{t-1}}}_{L_i},\underbrace{E^{t-1}_{i}}_{C_i},\underbrace{E^{t-1}_{(q+1)2^{t-1}-1}}_{R_i}\ ] \tag{18}$$

where $q = \lfloor i/2^{t-1}\rfloor$ and the equation i-th in $t-1$ stage is of the type:

$$E^{t-1}_{i} = \{a^{t-1}_{i}x_{q2^{t-1}-1} + b^{t-1}_{i}x_i + c^{t-1}_{i}x_{(q+1)2^{t-1}} = d^{t-1}_{i}\} \tag{19}$$

Figure 5 shows details of how each pair of triads ($[i]^{t-1}$ and $[j]^{t-1}$) are combined; each circle represents a reduction operation. First, the middle term of the equation R_i reduces the first term of L_j. The middle term of the new equation in L_j is used to reduce the final term of L_i and C_i. On the other hand, the final term in R_i reduces the middle term of the original L_j, and then, the new equation in L_j reduces the first term of R_j and C_j. At the end, both left equations will be identical (see L'); the same is true for both right equations (see R'). After n computation steps, the solution x_i can be immediately computed by dividing the second term of C_i by its independent term. This is the basic computation in the case of radix-2, but higher radix versions can be used. Therefore, each node operator element is a triad of equations that requires 3×16 bytes of storage. However, when dealing with adjacent equations, there is a property which means that the whole triad need not be stored, just a single equation, as the two others are easily obtained from adjacent equations. Specifically, the left and right equations are equal to two of the center equations. In step k, the left and right equations of $[i]$ can be obtained as follows:

$$L_i = C_a \rightarrow a = 2^k \times \lfloor i/2^k\rfloor$$
$$R_i = C_b \rightarrow b = 2^k \times (1+\lfloor i/2^k\rfloor)-1$$

Therefore, each element is represented by a single central equation and is stored in a *float*4 data type, since its right and left equations are easily obtained from the central equations of other elements. This property only arises in the first stage of the algorithm, where adjacent equations are stored in a common memory space; whereas in the remaining stages, triads need to be stored for each equation, since the central equations used for calculating the right and left equations could be placed into another memory

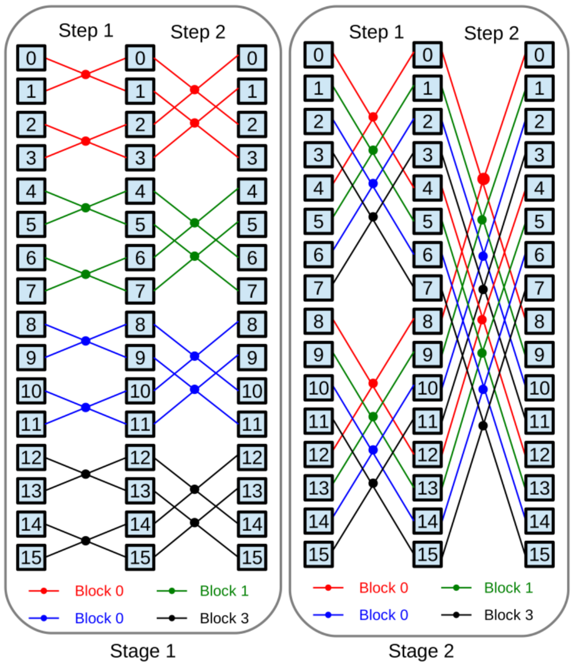


Fig. 6: Distribution of *ID-LD-TS* proposal with two stages for $N = 16$, $p = l = 1$ and $b_x = 2$

space. Henceforth, an element is formed by one equation in the first stage, thanks to the adjacent property, but by 3 equations (a triad) in the remaining stages.

Figure 6 shows the reason why this property cannot be applied in several stages. It shows the *MS-ID-TS* proposal for $n = 4$, $p = l = 1$ and $b_x = 2$, where each number-box represents an element. The computation is divided into two stages; the first stage processes the first and second steps, while the second stage performs the third and fourth steps, using four thread blocks in each step. In the first stage, adjacent elements are stored in the corresponding shared memory of each thread block, so only one equation per element is stored. However, it is easy to observe how this behavior changes in the second stage, as each thread block works with non-adjacent elements, storing the whole triads for each element. For example, element-4 is solved by block 0 in the second stage (although it was processed by block 1 in previous stage). Accordingly, the right equation of element 4, R_4, corresponds with the central equation of element 7, C_7, in the first step of the second stage. However, element-4 cannot access element-7, since they are in different thread blocks: element-4 is stored in thread block 0, whereas element-7 is contained in thread block 3. This fact forces us to store the corresponding three equations of each triad for all elements.

Changing the access pattern (i.e., changing the current data distribution among thread blocks) to another pattern where adjacent elements are placed together in the same thread block (working with portions of consecutive elements) would imply increasing the number of stages (with its corresponding latency penalty). This new communication pattern guarantees processing the maximum number of steps per stage, thus minimizing the number of stages. For non-first stages, we have preferred to launch a small number of kernels which store whole triads, instead of launching more kernels whose elements are single equations. We justify this decision on the basis that the new GPU architectures (and it is highly likely that future architectures too) increase their shared memory space, which is beneficial for this implementation.

Due to this limitation, it has been necessary to differentiate the s parameter depending on its being in the first stage or in subsequent ones. Thus s is split into s_1 and s_2, as data size is not the same for a single equation (first stage) as for a triad (remaining stages). Therefore, s_1 is used to represent the elements in the first stage, where it is not necessary to store triads, and s_2 is used in remaining stages, taking into account that elements are represented by triads of equations, $3 \times float4$. Note that both s_1 and s_2 refer to the number of stored elements, irrespective of their size. Thus, the first kernel's shared memory can hold more elements than other kernels' shared memory, as its elements are much lighter than the elements in the remaining stages, $s_1 > s_2$. In our proposal, the first stage computes $\lfloor s_1/p \rfloor$ steps and the remaining stages will compute $w = \lfloor \frac{(n-s_1)}{p \cdot (m-1)} \rfloor$ steps per stage. In order to take advantage of the data type used for representing the elements, s_1 should be as large as possible, since more steps can be performed in Stage 1 in comparison to other stages, while using the same amount of shared memory thanks to the adjacency property.

Moreover, instead of having only $s_2 = r^w$ elements per thread block in the remaining stages, our implementation stores s_2 elements of the same problem in each thread block, where the s_2 value is defined to maximize GPU parallelism, as explained below. Thus, each thread block computes several sets of r^w dependent elements from the same problem until s_2 is fulfilled. This increases warp parallelism, performing more work in each thread block. There is dependence among elements of the same set (computing w steps implies r^w elements), but sets are independent from each other. As each set operates separately without needing information from the other sets, the number of performed steps is still w, and all threads in a thread block working on the same set have the same l_y-identifier.

In order to better understand the mapping vector design, the CUDA implementation steps of our *MS-LD-TS* algorithm are analyzed below, as we have already done with the *MS-LD-FFT* algorithm. At the beginning of computation, there is a table which determines the number of stages and steps processed by stage (kernel) for each problem size. Likewise, there is another table which specifies s, l and, thus, the radix employed for each problem size.

1) Each thread loads P elements from global memory into registers. In the first stage, these elements are adjacent, benefiting coalescence. In the remaining stages, each thread loads triads of equations following the corresponding pattern.

2) Compute the first step using radix-P or a mixed radix.
3) Exchange data through shared memory. Equations are stored as *float4* elements in shared memory. Except in the first stage, there are three shared memory buffers, one for each triad.
4) Compute the following step.
5) If no step remains in that stage, then the result is written to global memory in last stage or triads are stored into global memory in the remaining stages.
6) There are two possibilities:
   a) If all the required stages have been completed, the algorithm ends.
   b) Otherwise, the next kernel reads triads from global memory, using the corresponding offset between their elements, restarting the process of this list from Point 1.

4.1 MS-ID-TS Mapping Vector

In our proposal, p is mapped to the lower part of the index-digits to ensure the global memory coalescence. Each component of the equation is stored in a different array, and each thread accesses four consecutive elements from up to four equations using $float4$ data type. In the first stage, the mapping vector of data on the GPU resources is as follows:

$$[\,\underbrace{t_{n+batch}\cdots t_{n+1}}_{b_y}\overbrace{\underbrace{t_n\cdots t_{l+p+1}}_{b_x}\;\underbrace{\underbrace{t_{l+p}\cdots t_{p+1}}_{l}\underbrace{t_p\cdots t_1}_{p}}_{s}}^{n}\,] \tag{20}$$

In order to determine the (p, s_1, l) tuple, two factors need to be considered: on the one hand, it is recommended to use the largest shared memory possible in order to compute the maximum number of steps in the first stage as explained above; on the other hand, it is also important to fulfill the three stated premises, analyzing each target architecture and finding a trade-off.

Following Premise 1, the number of stages should be minimized and is determined by s. In Kepler, $s_1 = 11$ implies only 1 active thread block and 25% of warp occupancy per SM; $s_1 = 10$ involves 3 active blocks and 38% of warp occupancy, whereas $s_1 = 9$ generates 6 active blocks and 38% of warp occupancy. Lower s_1 values underexploit shared memory, as registers would be the limiting factor of occupancy. Thus, $s_1 = 9$ is selected in order to achieve Premise 2, making it possible to solve $n \le 18$ problem sizes with only $m = 2$ stages. The same reasoning is applied to Maxwell, choosing $s_1 = 9$. This involves 8 active thread blocks and 32 active warps per SM. However, taking into account that s_2 stores at least $n - s_1$ elements, and each element occupies 48 bytes in the second stage, then this involves $s_2 \le 9$, owing to hardware limitations, and the second kernel occupancy would be very low when $n > 16$. In order to avoid this, $s_1 = 10$ is utilised when $n > 16$. Although some occupancy is lost in Stage 1, performance will be improved in the second stage, obtaining better global performance in the whole application. For example, note that executing $n = 17$ on Kepler with $s_1 = 9$ (and $p = 2$) in the first stage implies the following mapping vector in the second stage:

$$[\,t_{17+batch}\cdots t_{18}\overbrace{t_{17}\cdots t_{10}\;\underbrace{\underbrace{t_8\cdots t_3}_{l}\underbrace{t_2 t_1}_{p}}_{s_2=8}}^{n}\,] \tag{21}$$

Obtaining only 4 concurrent thread blocks and 8 active warps per SM in the second stage. Nevertheless, using $s_1 = 10$ and $s_2 = 7$ involves the following mapping vector:

$$[\,t_{17+batch}\cdots t_{18}\overbrace{t_{17}\cdots t_{11}\;\underbrace{\underbrace{t_7\cdots t_3}_{l}\underbrace{t_2 t_1}_{p}}_{s_2=7}}^{n}\,] \tag{22}$$

with 8 concurrent thread blocks and 8 active warps per SM in the second stage. Despite slightly reducing the number of concurrent thread blocks in the first stage, global performance is enhanced. Additionally, thanks to using $s_1 = 10$ in large problem sizes, up to $n \le 19$ sizes can be solved in only 2 stages. Maxwell provides up to 96 KB per SM, delaying this parameter update until $n > 17$, as it achieves a higher occupancy than Kepler at the same shared memory consumption. Once both s_1 and s_2 have been established, and taking into account that $p = 2$ according to [27], the following tuples are obtained for Kepler: $(p, s_1, l) = (2, 9, 7)$ when $n \le 16$, and $(p, s_1, l) = (2, 10, 8)$ when $16 < n \le 19$. Regarding Maxwell, these values are $(p, s_1, l) = (2, 9, 7)$ when $n \le 17$ and $(p, s_1, l) = (2, 10, 8)$ otherwise.

In the remaining stages, the mapping vector is:

$$\begin{aligned}&[\,\underbrace{t_{n+batch}\cdots t_{n+1}}_{b_y}\underbrace{t_n\cdots t_{b_x+l_y+p+1}}_{l_x}\\ &\underbrace{t_{b_x+l_y+p}\cdots t_{b_x+l_y+1}}_{p}\underbrace{t_{b_x+l_y}\cdots t_{b_x+1}}_{l_y}\underbrace{t_{b_x}\cdots t_1}_{b_x}\,]\end{aligned} \tag{23}$$

In this case, the use of p as of the $b_x + l_y + 1$ index-digits also ensures global memory coalescence. In the remaining stages, the performance parameters used are given by $p = 2$, $s_2 = max(6, n - s_1)$ and $(l_x, l_y) = ((n - s_1 - p), (s_2 - l_x - p))$. In the case of $s_2 = 6$, this ensures having 3072 shared memory bytes per block, not limiting block parallelism in either Kepler or Maxwell architectures and executing several sets per block, in the case of short problem sizes. When $n > (6 + s_1)$, a single set of elements is processed by each block, consuming as much shared memory as necessary. Regarding the distribution of threads in each thread block, $l_x = (n - s_1 - p)$, L_x of them collaborate in the same set of equations that have dependencies between each other, as explained above, while $l_y = (s_2 - l_x - p)$ represents the fact that there are L_y sets of equations of the same problem being independently solved in the thread block. Therefore, our implementation uses L_x for working on the same set, whereas L_y sets of the same problem are solved in parallel in that block. Finally, B_x thread blocks work on the same problem, whereas B_y blocks work on

	Platform 1, Platform 2	*Platform 3*
CPU	Intel Xeon E5-2660 2.2 GHz	Intel Core i7-2600 3.4 GHz
Memory	64 GB DDR3 1600	8 GB DDR3 1333
OS	CentOS 6.4	Ubuntu 12.04 LTS
Compiler	GCC 4.4.7	GCC 4.6.3
GPU	*Nvidia Tesla K20, Nvidia Tesla K40*	*Nvidia GeForce GTX980*
Driver	340.58, SDK 6.0	343.22, SDK 6.5

TABLE 3: Description of the test platforms

different problems in batch mode.

The string operator used in the first kernel is as follows:

$$\prod_{i=1}^{s_1/p-1}\left[\Upsilon_1^p\Gamma^p_{(i+1)\cdot p,i\cdot p+1,p,1}\right]\Upsilon_1^p\Gamma^p_{s_1,1} \tag{24}$$

In the case of $(s_1\ mod\ p \neq 0)$, an extra step is needed. For the remaining kernels, the same expression is used but an offset of $l_y + b_x + 1$ digits is applied in sub-indexes, as its mapping vector is different, executing $\frac{n-s_1}{p}$ steps.

For example, for $n = 14$, the data mapping vector in the first stage would be

$$[\ \underbrace{\cdots t_{15}}_{b_y}\ \overbrace{\underbrace{t_{14}t_{13}t_{12}t_{11}}_{b_x}\ \underbrace{t_{10}t_9t_8t_7t_6t_5t_4t_3}_{l}\ \underbrace{t_2t_1}_{p}}^{n}\ ] \tag{25}$$

and in the second stage

$$[\ \underbrace{\cdots t_{15}}_{b_y}\overbrace{\underbrace{t_{14}t_{13}t_{12}t_{11}t_{10}}_{l_x}\ \underbrace{t_9t_8}_{p}\ \underbrace{t_7t_6}_{l_y}\underbrace{t_5t_4t_3t_2t_1}_{b_x}}^{n}\ ] \tag{26}$$

5 Experimental Results

In this section, the results of both the FFT algorithm and the tridiagonal system solver are presented and analyzed. In both cases, test data are already on the GPU, thus there are no data transfers during the benchmarks. The experiments are run in single precision. Table 3 describes the test platforms used in our experiments. Platform 1 and 2 share similar features, presenting a Kepler GPU architecture, whereas Platform 3 has a Maxwell GPU architecture. The Kepler K20 card has 13 SMX with 192 CUDA cores each one, whereas Kepler K40 has 15 SMX with 192 CUDA cores and Maxwell GTX980 has 16 SMM with 128 CUDA cores.

5.1 FFT Results

FFT performance is commonly expressed in GFlops. In the case of the complex FFT, the GFlop rate is given by the formula $5N \cdot log_2(N) \cdot batch \cdot 10^{-9}/t$ where *batch* is the total amount of data sequences being processed and t is the time (seconds). In order to determine the number of batches for each N problem, the total amount of data to be processed is fixed at 2^{24} complex elements, so the number of batch problems depends on the problem size ($batch = 2^{24}/N$).

Table 4 presents the performance and profiler analysis for the different *MS-ID-FFT* versions on Platform 1, highlighting the best result for each problem size. The first column indicates the different version being used: *MS-ID-FFT.V1* with $p = 4$ and *MS-ID-FFT.V2* with $p = 3$. A different number of results are shown in each column, depending on the number of kernels executed. If an entry shows a single column value for different kernels, this means all kernels share the same result. Regarding the kernel profiler analysis, tables have detailed information on the block size (L), the number of registers per thread, the amount of shared memory per thread block and the achieved SM occupancy. It should be noted that the maximum number of registers for all cases is 56, thus there is no local memory spilling. Note that in all cases, the amount of available shared memory is a limiting factor. Moreover, the number of registers becomes a constraint on these thread block sizes, since more than 32 registers per thread reduces the number of concurrent blocks per SM.

For $n \leq 16$, *MS-ID-FFT.V1* achieves better results, even with only 25% of the maximum SM occupancy (38% on Maxwell architectures). It launches a single kernel for $n = 12$, obtaining 504.68 GFLOPS. This performance drops, to 345.12 GFLOPS, when $n = 13$ owing to the launching of two kernels due to the amount of available shared memory becoming a limiting factor on a single kernel (17408 bytes when $n = 12$). Although *MS-ID-FFT.V2* has better SM occupancy due to a lower register/shared memory consumption, better results are attained with *MS-ID-FFT.V1* thanks to the reduction in computational steps. For example, with $n = 12$, 3 computational steps are performed in the first kernel, and 1 in the second one using *MS-ID-FFT.V1*; whereas 4 computational steps with the first kernel and 1 in the second one with *MS-ID-FFT.V2*. Nonetheless, *MS-ID-FFT.V1* uses 3 kernels from $n = 17$, reducing performance considerably. At this point, *MS-ID-FFT.V2* shows better results as it still utilizes two kernels owing to its lower shared memory consumption: 444.55 GFLOPs for $n = 17$ as opposed to 301.54 GFLOPs for *MS-ID-FFT.V1* for the same n. From $n = 19$, both versions launch three kernels and *MS-ID-FFT.V1* once again exhibits better performance thanks to its fewer computing steps. An analogous analysis is performed for Platform 3 in Table 5, obtaining similar results than Table 4 about the FFT version to be chosen for each n.

The performance results of our FFT algorithm are shown in Figure 7 for each test platform, comparing the results with the CUFFT (included in NVIDIA's SDK). Both FFT algorithms are highly optimized to take advantage of GPU computing resources and perform most data exchanges in shared memory. However, as the number of kernels increases, the global memory bandwidth becomes the main limiting factor. In a previous work [27], the performance of CUFFT is proved with respect to other libraries, such as the Nukada library.

The graphs show that our best results are obtained by combining both FFT versions, thanks to a tuning process that generates a table which selects the best version for each problem. The graphs show a number of spikes before $n = 13$ and $n = 19$. The reason for this behavior is the invocation of an extra kernel. After $n = 12$, the algorithm

	n	GFLOPS	L	Reg	Shared mem. (bytes)	Occup. (%)	kernels
V1	12	**504.68**	256	52	17408	25	1
	13	**345.12**	256	51	16448,17408	25	2
	14	**372.16**	256	51	16448,17408	25	2
	15	**397.96**	256	52	16448,17408	25	2
	16	**424.87**	256	51	16448,17408	25	2
	17	301.54	256	47,53,53	16448,17408,17408	25	3
	18	316.24	256	47,53,53	16448,17408,17408	25	3
	19	**327.02**	256	47,53,54	16448,17408,17408	25	3
	20	**344.75**	256	47,53,52	16448,17408,17408	25	3
	21	**360.76**	256	50,56,53	16448,17408,17408	25	3
	22	**380.34**	256	50,56,53	16448,17408,17408	25	3
	23	**396.41**	256	50,56,54	16448,17408,17408	25	3
	24	**414.03**	256	50,56,53	16448,17408,17408	25	3
V2	12	504.08	256	52	17408	25	1
	13	345.07	64,512	35	2336,16640	48	2
	14	370.94	64,512	36	2336,16640	48	2
	15	393.2	64,512	36	2336, 16640	48	2
	16	412.74	512	36	16640	48	2
	17	**444.55**	512	50,37	16640	25,48	2
	18	**459.08**	512	36	16640	48	2
	19	319.24	256,512,512	47,38,41	16640	25,48,48	3
	20	334.54	256,512,512	47,38,37	16640	25,48,48	3
	21	330.92	256,256,512	47,53,37	16640,17408,16640	25,25,48	3
	22	302.84	256,512,512	47,38,37	16640	25,48,48	3
	23	329.03	256,256,512	50,56,37	16640,17408,16640	25,25,48	3
	24	336.77	256,256,512	50,56,38	16640,17408,16640	25,25,48	3

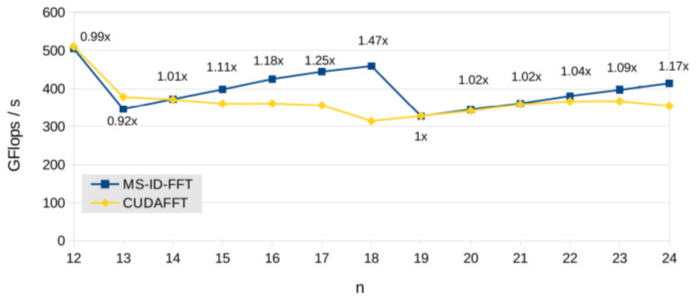
TABLE 4: Complex MS-ID-FFT kernel performance and profiler analysis on Platform 1

	n	GFLOPS	L	Reg	Shared mem. (bytes)	Occup. (%)	kernels
V1	12	**615.37**	256	50	17408	38	1
	13	**324.3**	256	50	16448,17408	38	2
	14	**351.86**	256	53	16448,17408	38	2
	15	**372.45**	256	53	16448,17408	38	2
	16	**396.03**	256	50	16448,17408	38	2
	17	281.94	256	46,51,52	16448,17408,17408	75	3
	18	295.62	256	46,51,56	16448,17408,17408	38	3
	19	**310.45**	256	46,51,56	16448,17408,17408	38	3
	20	**322.78**	256	46,51,50	16448,17408,17408	38	3
	21	**333.83**	256	48,54,52	16448,17408,17408	38	3
	22	**347.55**	256	48,54,56	16448,17408,17408	38	3
	23	**366.98**	256	48,54,52	16448,17408,17408	38	3
	24	**382.31**	256	53,54,53	16448,17408,17408	38	3
V2	12	612.76	256	50	17408	38	1
	13	313.6	64,512	35	2336,16640	75	2
	14	334.854	64,512	35	2336,16640	75	2
	15	362.41	64,512	35	2336, 16640	75	2
	16	360.62	512	35	16640	75	2
	17	**392.22**	512	48,36	16640	50,75	2
	18	**394.83**	512	36	16640	75	2
	19	295.23	256,512,512	46,35,36	16640	50,75,75	3
	20	306.6	256,512,512	46,35,36	16640	50,75,75	3
	21	312.29	256,256,512	48,51,37	16640,17408,16640	50,50,75	3
	22	291.28	256,512,512	48,35,37	16640	50,75,75	3
	23	327.53	256,256,512	48,54,36	16640,17408,16640	50,50,75	3
	24	345.4	256,256,512	48,54,37	16640,17408,16640	50,50,75	3

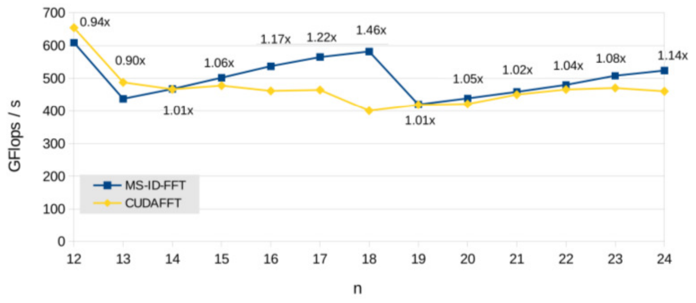
TABLE 5: Complex MS-ID-FFT kernel performance and profiler analysis (Platform 3)

is executed with two kernels instead of a single one. This change is required due to shared memory constraints. For $n = 13$, our library would require 8192 elements using 8 bytes per complex element, therefore 64 KB of shared memory (without considering padding space to reduce bank conflicts), which exceeds the maximum allowed. Even if we divided the exchange of complex data (real and imaginary parts), the GPU occupancy would be quite low. As can be observed in the graphs, a similar problem occurs when transitioning from $n = 18$ to $n = 19$, which requires a third kernel to process the problem. Thus, the global memory requirements are doubled and the performance is reduced proportionally. CUFFT also shows this behavior in its results for the three platforms, but ocurring with different problem size transitions. Each library adjusts them depending on its distribution of GPU resources and strategy. Changing the number of kernels always implies an impact on performance.

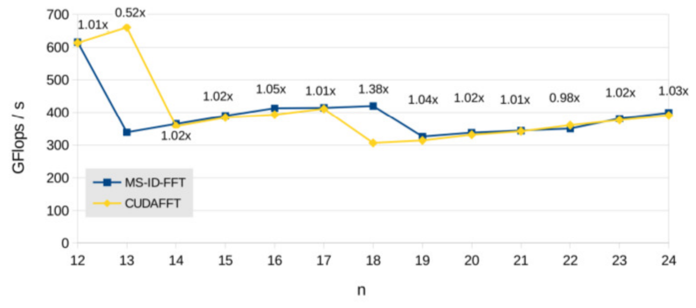
On Platform 1, our library performs better than CUFFT, as can be observed, with an improvement of up to 1.47x, being 1.09x on average. This large performance gap is mostly linked to the difference in the number of kernel passes required by the kernels (for $n = 18$ CUFFT uses three kernels while MS-ID-FFT requires only two). In most cases, our algorithm is able to maximize global memory bandwidth utilization and split the computation at the optimal point to distribute the workload evenly among the kernels. On Platform 2, results are very similar to Platform 1, obtaining an improvement of up to 1.46x. On Platform 3, our library surpasses CUFFT, except in $n = 13$ and $n = 22$. It should be pointed out that CUFFT uses one kernel for $n = 13$, whereas our proposal uses 2 kernels, adversely affecting performance. However, our proposal launches only two kernels when $n = 18$, surpassing CUFFT which launches three kernels. For $n = 18$, our library is 1.38x faster, but for other sizes, the speed-up is more modest compared to the Kepler architecture, being 1.01x on average. The reason behind this behavior is the ratio between computing power



(a) Platform 1



(b) Platform 2



(c) Platform 3

Fig. 7: Performance comparison of Complex $MS-ID-FFT$ proposal

and memory bandwidth in Maxwell. The Maxwell architecture was designed with efficiency in mind, employing larger caches that usually reduce bandwidth requirements, when data is reused. However, the FFT algorithm needs to read all data in each kernel pass. On the other hand, the amount of shared memory is also increased, which results in greater GPU occupancy and more efficient data exchanges, therefore reducing the computation but forcing the kernel to wait for data more often. As can be observed, this issue affects both MS-ID-FFT and CUFFT. Although the speed-up achieved is not as impressive as the results for tridiagonal system solvers, we would like to stress the fact that our library, based on a methodology, outperforms CUFFT, a fully optimized and well-known library for FFT on GPUs.

5.2 Tridiagonal System Solver Results

In the case of tridiagonal system solver, performance is measured in million rows processed per second, MROWS/s. Therefore, the MROWS/s value is obtained using the expression $N \cdot r^{batch} \cdot 10^{-6}/t$. In these tests, data are initialized using diagonally dominant equation systems.

Our *MS-ID-TS* proposal considers problem sizes with $8 \leq n \leq 19$. For dealing with larger sizes, more GPU memory would need to be used (i.e. several GPU devices may need to be used). In order to compute small problem sizes $n < 8$, it is preferable to use the approach obtained in [27], optimized for problems that fit directly in shared memory, exchanging triads during computation without resorting to global memory.

Table 6 and Table 7 present the profiler analysis for Platform 1 and Platform 3, respectively. Each column contains the values obtained for the two executed kernels. Firstly, it should be noted that the $(p, s_1, l) = (2, 9, 7)$ tuple was established on Kepler in Section 4 when $n \leq 16$, as can be seen in Table 6, obtaining 768.6 MRows/s when $n = 16$. From $n = 17$, the tuple $(p, s_1, l) = (2, 10, 8)$ is used for the reasons explained in Section 4 (decreasing block parallelism on kernel 1, but increasing global performance since block parallelism is increased on kernel 2), achieving 664.2 MRows/s. Table 6 shows global performance for both cases, $s_1 = 9$ and $s_1 = 10$, highlighting the best result. However, Table 7 shows 1554.2 MRows/s when $n = 16$ and 1623.4 MRows/s when $n = 17$. In this case, the same $(p, s_1, l) = (2, 9, 7)$ tuple is being used for both cases, as Maxwell architecture provides up to 96 KB per SM, delaying $s_1 = 10$ until $n = 17$, as it keeps a higher occupancy than Kepler at the same shared memory consumption. With $n = 18$, 1556 MRows/s are achieved in Maxwell platform, since the new $(p, s_1, l) = (2, 10, 8)$ tuple is being employed, achieving 1.29x with respect to the $s_1 = 9$ implementation. Finally, it should be observed that greater occupancies are achieved on Maxwell architectures than on Kepler architectures at the same level of shared memory consumption due to the increased SM shared memory size in Maxwell, as explained above.

Figure 8 shows the results of executing a single problem (solid lines) and multiple batches (dashed lines)

n	MRows/s	L	Reg	Shared mem. (bytes)	Occup. (%)
12	690.6	128,16	56,66	8192,3072	36,25
13	826.2	128,16	56,66	8192,3072	36,25
14	888.4	128,16	56,66	8192,3072	36,25
15	792.6	128,16	56,66	8192,3072	36,25
16	768.6	128,32	56,68	8192,6144	36,13
17	660.3	128 ,64	56,68	8192,12288	36,13
	664.2	256,32	59,68	16384,6144	36,13
18	645	128,128	56, 68	8192,24576	36,13
	687	256,64	59,66	16384,12288	36,13
19	620	256,128	59,61	16384,24576	36,13

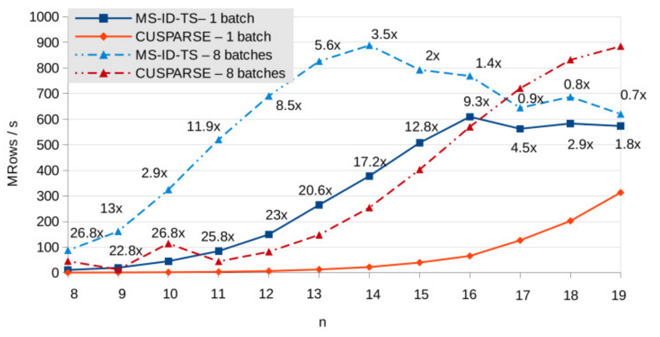
TABLE 6: Complex MS-ID-TS kernel performance and profiler analysis (Platform 1)

n	MRows/s	L	Reg	Shared mem. (bytes)	Occup. (%)
12	1202.1	128,16	55,63	8192,3072	48,33
13	1130.9	128,16	55,63	8192,3072	48,33
14	1349.3	128,16	55,63	8192,3072	48,33
15	1508.8	128,16	55,63	8192,3072	48,33
16	1554.2	128,32	55,64	8192,6144	48,16
17	**1623.4**	128,64	55,65	8192,12288	48,16
	1516	256,64	59,59	16384,12288	48,16
18	1204	128,128	55,59	8192,24576	48,16
	1556	256,64	59,65	16384,12288	48,16
19	1250.4	256,128	59,63	16384,24576	48,13

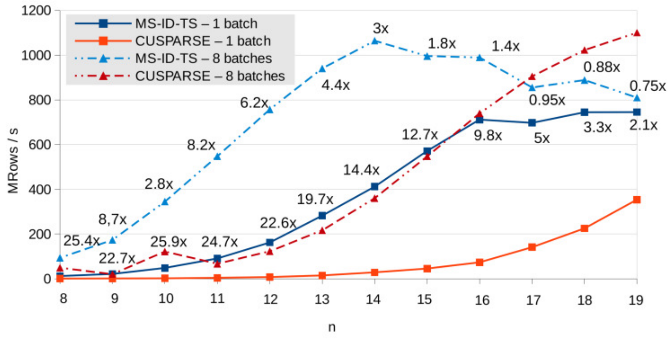
TABLE 7: Complex MS-ID-TS kernel performance and profiler analysis (Platform 3)

on Platforms 1, 2 and 3. The performance comparison with respect to the CUSPARSE library for one batch is very similar on all three platforms. The performance growth of CUSPARSE is very slow, as NVIDIA launches 10 kernels. However, the performance growth in our solver is immediate. From $n = 16$ on Platform 1 and Platform 2; and from $n = 17$ on Platform 3, performance begins to decrease, as was expected owing to the replacement of $s_1 = 9$ by $s_1 = 10$. In Section 4 and Tables 6 and 7, we have justified the peaks in $n = 16$ (on Kepler) and $n = 17$ (on Maxwell). As having more elements requires more shared memory, then the fixed amount of shared memory, optimized in our implementation, is not sufficient and needs to be increased. This increase in shared memory leads to reduced occupancy and a loss of performance, obtaining those peaks. In the case of eight batches, the speedup with respect to CUSPARSE is more modest, as storing triads from all batches in global memory consumes much more bandwidth. As more batches are introduced, more GPU parallelism is exploited. Furthermore, more global memory operations are issued, and L1/L2 cache behavior will determine the location of peaks for each batch execution. In all cases, the occupancy in the second kernel is lower, especially when dealing with large problem sizes, where the shared memory becomes a limiting factor. As future work, we are considering implementing a hybrid algorithm that does not use the Wang and Mou approach in the second stage in order to reduce memory use.

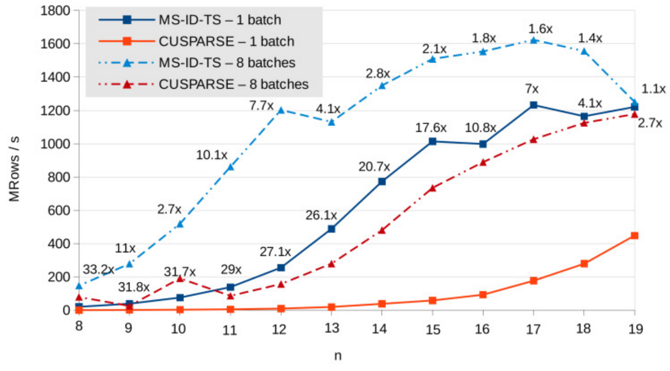
In the case of Platform 1 and one batch, our solver obtains up to 26.8x improvement over CUSPARSE, being



(a) Platform 1



(b) Platform 2



(c) Platform 3

Fig. 8: Performance comparison of $MS-ID-TS$ proposal

16.37x times faster on average. With eight batches, this proposal is, on average, 4.41x faster. On Platform 2, it is up to 25.9x faster for one batch, 15.7x on average; whereas it provides up to 8.7x of speedup when processing eight batches simultaneously. Additionally, up to 33.2x of improvement is achieved with one batch on Platform 3, 20.14x on average, while it obtains up to 11x of speedup with eight batches, 4x on average.

6 Conclusions

This paper presents a new multi-stage strategy that permits the tuning of ID-algorithms which can be represented according to permutations of the digits of its elements and for large problem sizes which fit into global memory of a single GPU. This strategy is based on the use of a tuned mapping vector following three Premises: the minimization of the number of stages, the balancing between warp and block parallelism, and an increase of the computational load per thread.

This methodology makes it possible to create efficient parallel algorithms for GPU architectures with little effort, taking into account only three parameters (n, p, b_x). At compile-time, the tuned kernels generated are obtained with index-digit permutations and achieve competitive performance. Specifically, in this work we have proposed two different proposals for FFT and tridiagonal system solver algorithms, based on the strategy outlined herein, obtaining two efficient algorithms: *MS-ID-FFT* and *MS-ID-TS*. The performance of our proposals has been analyzed and compared to other well-known libraries. *MS-ID-FFT* shows an improvement of up to 1.47x on Platform 1 (Kepler), up to 1.46x on Platform 2 (Kepler) and up to 1.38x on Platform 3 (Maxwell) with respect to CUDAFFT library. *MS-ID-TS* is also up to 26.8x faster on Platform 1, up to 25.9x on Platform 2 and up to 33.2x on Platform 3 with respect to CUSPARSE.

Our future work will be aimed at applying the methodology presented to other ID problems, such as parallel prefix scan operation and sorting operators. On the other hand, we also plan to extend the methodology for problem sizes that are too large for global memory.

References

[1] D. Fraser, "Array Permutation by Index-Digit Permutation," *Journal of ACM*, vol. 23, no. 2, pp. 298–309, 1976.
[2] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.
[3] Y. Dotsenko, S.S. Baghsorkhi, B. Lloyd and N.K. Govindaraju, "Auto-Tuning of Fast Fourier Transform on Graphics Processors," in *Proc. of Principles and Practice of Parallel Programming (PPoPP '11) (2011)*, 2011, pp. 257–266.
[4] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT Library for CUDA GPUs," in *Proc. of the Conf. on High Perf. Computing Networking, Storage and Analysis (SC'09) (2009)*, 2009, pp. 1–10.
[5] A. Nukada, K. Sato, and S. Matsuoka, "Scalable Multi-GPU 3-D FFT for TSUBAME 2.0 Supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 44:1–44:10.
[6] J. Park, G. Bikshandi, K. Vaidyanathan, P. T. P. Tang, P. Dubey, and D. Kim, "Tera-scale 1D FFT with Low-communication Algorithm and Intel&Reg Xeon Phi&Trade Coprocessors," in *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13 (2013), 2013, pp. 34:1–34:12.
[7] D. Takahashi, "Implementation of Parallel 1-D FFT on GPU Clusters," in *2013 IEEE 16th International Conference on Computational Science and Engineering*, 2013, pp. 174–180.
[8] C. Wang, S. Chandrasekaran, and B. Chapman, "cusFFT: A High-Performance Sparse Fast Fourier Transform Algorithm on GPUs," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 963–972.
[9] *CUDA CUFFT Library*, NVIDIA, 2012, v5.0. [Online]. Available: https://developer.nvidia.com/cufft
[10] Y. Zhang, J. Cohen, J.D. Owens, "Fast Tridiagonal Solvers on the GPU," in *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010, pp. 127–136.
[11] A. Davison and J. D. Owens, "Register Packing for Cyclic Reduction: A Case Study," in *Proc. of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 4:1–4:6.
[12] A. Davidson, Y. Zhang and J.D. Owens, "An Auto-tuned Method for Solving Large Tridiagonal Systems on the GPU," in *Proc. of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2011)*, 2011, pp. 956–965.

[13] R. Hockney and C. Jesshope, *Parallel Computers 2: Architecture, Programming and Algorithms*. Taylor & Francis, 1988.

[14] L. H. Thomas, "Elliptic Problems in Linear Difference Equations over a Network," *Watson Sci. Comput. Lab. Rep., Columbia University*, 1949.

[15] F. Arguello, D. Heras, M. Boo, and J. Lamas-Rodriguez, "The Split-and-Merge Method in General Purpose Computation on GPUs," *Parallel Computing*, vol. 38, no. 67, pp. 277 – 288, 2012.

[16] R. W. Hockney, "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis," *Journal of ACM*, vol. 12, no. 1, pp. 95–113, 1965.

[17] L.-W. Chang and W.-W. Hwu, "Mapping tridiagonal solvers to linear recurrences," *Technical Report, University of Illinois at Urbana-Champaign*, 2013.

[18] H.-S. Kim, S. Wu, L.-W. Chang, W.W. Hwu, "A Scalable Tridiagonal Solver for GPU," in *Procd. of Int. Conf. on Parallel Processing (2011)*, 2011, pp. 444–453.

[19] A. H. Sameh and D. J. Kuck, "On stable parallel linear system solvers," *J. ACM*, vol. 25, no. 1, pp. 81–91, 1978.

[20] L.-W. Chang, J.A. Stratton, H.-S. Kim, W. W. Hwu, "A Scalable, Numerically Stable, High-performance Tridiagonal Solver Using GPUs," in *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12) (2012)*, 2012, pp. 27:1–27:11.

[21] I. Venetis, A. Kouris, A. Sobczyk, E. Gallopoulos, and A. Sameh, "A Direct Tridiagonal Solver based on Givens Rotations for GPU Architectures," *Parallel Computing*, vol. 49, pp. 101 – 116, 2015.

[22] D. Zhao and J. Yu, "Efficiently Solving Tri-diagonal System by Chunked Cyclic Reduction and single-GPU Shared Memory," *J. of Supercomputing*, vol. 71, no. 2, pp. 369–390, 2015.

[23] *CUDA CUSPARSE Library*, NVIDIA, Aug. 2012, v5.0. [Online]. Available: https://developer.nvidia.com/cusparse

[24] E. László, M. Giles, and J. Appleyard, "Manycore Algorithms for Batch Scalar and Block Tridiagonal Solvers," *ACM Trans. Math. Softw.*, vol. 42, no. 4, pp. 31:1–31:36, 2016.

[25] X. Wang, W. Xue, J. Zhai, Y. Xu, W. Zheng, and H. Lin, "A Fast Tridiagonal Solver for Intel MIC Architecture," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 172–181.

[26] K. Li, W. Yang, and K. Li, "A Hybrid Parallel Solving Algorithm on GPU for Quasi-Tridiagonal System of Linear Equations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2795–2808, 2016.

[27] J. Lobeiras, M. Amor and R. Doallo, "Designing Efficient Index-Digit Algorithms for CUDA GPU Architecture," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1331–1343, 2016.

[28] F. Argüello, M. Amor and E.L. Zapata, "FFTs on Mesh Connected Computers," *Parallel Computing*, vol. 22, no. 1, pp. 19–38, 1996.

[29] J. Lobeiras, M. Amor and R. Doallo, "BPLG: A Tuned Butterfly Processing Library for GPU Architectures," *International Journal of Parallel Programming*, vol. 43, no. 6, pp. 1078–1102, 2015.

[30] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd ed. Morgan Kaufmann, 2012.

[31] Y. Yang, P. Xiang, M. Mantor, N. Rubin and H. Zhou, "Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput," in *Proc. of the 21st Int. Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12, 2012, pp. 283–292.

[32] Y. Yang and H. Zhou, "CUDA-NP: realizing nested thread-level parallelism in GPGPU applications," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP'14, (2014)*, 2014, pp. 93–106.

[33] K. Gupta, J. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Proceedings of Innovative Parallel Computing*, ser. InPar '12 (2012), 2012.

[34] *CUB Library*, Nvidia Comp.., 2015. [Online]. Available: http://nvlabs.github.io/cub

[35] Y.-Ch. Lin and L.-L. Hung, "Fast problem-size-independent parallel prefix circuits," *Journal Parallel Distributed Computing*, pp. 382–388, 2009.

[36] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU technology conference, GTC*, vol. 10. San Jose, CA, 2010, p. 16.

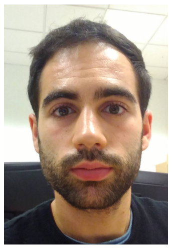
[37] V. Volkov, "Use Registers and Multiple Outputs per Thread on GPU," in *6th International Workshop on Parallel Matrix Algorithms and Applications*, 2010.

[38] A. P. Dieguez, M. Amor, J. Lobeiras and R. Doallo, "Operator String Algebraic Properties and Usage," *Internal Report at University of A Coruña*, 2016. [Online]. Available: {http://gac.des.udc.es/~aperezdieguez/AnnexA5.pdf}

[39] T.G. Stockham, "High-Speed Convolution and Correlation," in *Proc. of the Spring joint computer conference*, 1966, pp. 229–233.

[40] X. Wang and Z.G. Mou, "A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers," in *Proc. of the Third IEEE Symposium on Parallel and Distributed Processing (1991)*, 1991, pp. 810–817.

[41] J. L. Pey, "Design and evaluation of tridiagonal solvers for vector and parallel computers," Ph.D. dissertation, Universitat Politecnica de Catalunya, 1995.



Adrián P. Diéguez obtained his Bachelor of Science and Master of Sience at University of A Coruña, Spain, in 2013 and 2014, respectively, and he is currently a PhD candidate. He joined the Computer Architecture Research Group as a researcher in 2013. His main research interests are parallel algorithms and computer graphics architectures.

Margarita Amor holds BSc and PhD degrees in Physics from the University of Santiago de Compostela, Spain (1993 and 1997 respectively). She is currently an associate professor at the Department of Electronic and Systems, at University of A Coruña. Her research interests include the areas of computer graphics and parallel computing.

Jacobo Lobeiras holds BSc and PhD degrees in Computer Science from the University of A Coruña, Spain (2010 and 2014 respectively). His main research topics are GPU computing and signal processing. His research interests are mainly focused on the areas of computer graphics and parallel computing.

Ramón Doallo received his Ph.D in Physics from the Univ. Santiago de Compostela. He is Full Professor and Head of the Computer Architecture Research Group at University of A Coruña. He has 28 years of experience in research and development in the area of High-Performance Computing (HPC), covering a wide range of topics such as parallel and distributed algorithms and applications, cloud computing, Big Data processing, processor architecture, and computer graphics. He has published more than 200 technical papers on these topics.