Enhancing Data Center Performance with GPU-Accelerated Dynamic Programming Algorithms

Andreu Roca Serrano

February 8, 2024

Abstract– Data generation rates for problems involving dynamic programming (DP) algorithms have grown exponentially. The inability of classical CPU architectures to cope with the ever-increasing amount of data has led Data Processing Centers (DPCs) to incorporate accelerators that can perform thousands of operations in parallel. Among the available options, GPUs have emerged as the preferred choice due to their widespread availability, ease of programmability, and attractive price-to-performance ratio.
In this TFG, we study the suitability of the new NVIDIA GPU Hopper architecture as an effective solution for DPCs solving big-data problems that involve dynamic programming. These problems span diverse domains, including personalized medicine, genomics, systems security, signal processing, and artificial intelligence. As a case study, we use the state-of-the-art supercomputer Marenostrum 5 from the Barcelona Supercomputing Center (BSC).
As a result of this study, we evaluated the potential of the H100 GPU combined with the new DPXs instructions and their suitability to aid in dynamic programming algorithms.

Keywords– Dynamic Programming, GPU, Data Processing, Performance Analysis

1 Introduction

Data size of problems requiring dynamic programming (DP) algorithms has been growing exponentially in recent years. Areas that make extensive use of DP algorithms include computational biology (sequence alignment) [1]- [7], artificial intelligence (natural language processing) [8], security (virus signature matching, binary diffing, networks or spam filtering) [9] [10], signal processing (speech recognition) [11], and software development (source code diffing) [12] among others [13]-[21].

All of these technologies that revolve around DP need efficient ways to process the generated data which is being massively produced, surpassing Moore's law. The solution that is being adopted by modern data centers is using accelerators (GPUs, FPGAs, ASICs) to cope with the increasing data production rate.

There is a trend to add specialized hardware to general-purpose devices to accelerate compute-intensive applications. For instance, the rise of artificial intelligence (AI) generated such a computational need that NVIDIA introduced specialized hardware for AI applications: the tensor cores. Now, with the need that arose from the use of DP algorithms, Nvidia introduced an extension of the GPU instruction set architecture (ISA) that performs basic operations of the DP algorithms: the DPX instructions.

This work presents a study of the effect that new GPU architectures (i.e., the NVIDIA H100 GPU) have on data processing centers (DPCs) and their ability to process big data faster and more efficiently.

The present document is structured as follows. Firstly, an analysis of the state of the art and the need for computing DP algorithms efficiently. Secondly, an analysis of two DP algorithms. Thirdly, an overview of GPU programming, the Hopper architecture, DPX and CUDA. Fourthly, an explanation of the roofline models and some considerations about performance. Fifthly, the experimental set-up and the metrics used. Sixthly, the results obtained in the static analysis and the different benchmarks performed. Finally, the conclusions drawn from the study and the consideration of future lines.

2 State of the Art

Nowadays, CPUs are good for general-purpose low-latency low-throughput computation but have proven not to scale consistently enough to deal with massively parallel problems. That has led high-performance computing (HPC)

• E-mail de contacte: andreurocaserrano@gmail.com
• Mencio realitzada: Tecnologies de la Informació
• Treball tutoritzat per: Juan Carlos Moure i Quim Aguado (Arquitectura de computadors)
• Curs 2023/24

data centers to rely on GPUs as accelerators for throughput-oriented computation.

The usage of custom specialized hardware for solving DP problems has also been considered, some examples are GenDP [22], EXMA [23], and Genesis [24], which achieve high efficiency at the cost of not being general purpose. Nowadays, manufacturers are introducing specialized hardware into general-purpose GPUs. Specifically, NVIDIA has introduced the DPX instructions for the acceleration dynamic programming algorithms.

Due to the DPX instructions being so recent, only one implementation uses them: CudaSW++4.0 [25], a protein alignment program that accelerates the Smith-Waterman algorithm.

3 Background

3.1 Dynamic programming

Dynamic programming (DP) is an algorithmic optimization technique that simplifies a complicated problem by breaking it down into simpler overlapped sub-problems and storing and reusing the partial results to avoid recomputation. In this study, we analyze two DP algorithms: Needleman-Wunsch (NW), which is used for sequence alignment, and dynamic time warping (DTW), which is used for signal analysis.

3.1.1 Needleman-Wunsch algorithm

Needleman-Wunsch is used to find the optimal alignment between two sequences. It finds the minimum transformations needed to convert one sequence into the other. Consider the sequences *T* (target), of length *m*, where T_j is the base *j* in the sequence *T*. And sequence *Q* (query), of length *n*, where Q_i is the base *i* in the sequence *Q*. It is possible to store the score for each partial alignment in a matrix of dimensions $n \times m$.

Consider a set of penalties $\{C, X, I, D\}$ that represents the cost of a match, mismatch, insertion, and deletion operation, respectively. We define a matrix M of size $(n+1) \times (m+1)$, where $M_{i,j}$ represents the cell of row i and column j. Eq.1 shows the recurrences to fill the matrix M.

$$M_{i,j} = \begin{cases} j \cdot I & if \quad i = 0 \\ i \cdot D & if \quad j = 0 \\ \min \begin{cases} M_{i-1,j-1} + \delta(T_j, Q_i) \\ M_{i-1,j} + D \\ M_{i,j-1} + I \end{cases} & if \quad i,j > 0 \end{cases} \tag{1a}$$

$$\delta(T_j, Q_i) = \begin{cases} C & if \quad T_j = Q_i \\ X & if \quad T_j \neq Q_i \end{cases} \tag{1b}$$

The first row and column do not contain information or results, those values are used to initialize the matrix and deal with border conditions.

To find the optimal alignment path (i.e., the minimum number of operations that convert one sequence into the other), we need to perform a traceback. The traceback process traverses the matrix M starting from the bottom-right cell ($M_{n+1,m+1}$) up to the top-left cell ($M_{0,0}$), which requires storing the whole matrix in memory. If we only want to know the cost of aligning the sequences (without knowing the optimal alignment path), it is enough to just have two rows, or two columns, residing in memory.

3.1.2 Dynamic Time Warping algorithm

Dynamic time warping is used to compare signals and determine their similarity without depending on time. Again, we define two signals *T* (target) of length *m*, and *Q* (query) of length *n*, and define a matrix M of size $n+1 \times m+1$, where $M_{i,j}$ representing the cell of row i and column j. The algorithm recurrences is shown in Equation 2.

$$M_{i,j} = \begin{cases} 0 & if \quad i,j = 0 \\ \infty & if \quad i = 0 \neq j \\ \infty & if \quad j = 0 \neq i \\ \mathcal{D}(T_j, Q_i) + \min \begin{cases} M_{i-1,j-1} \\ M_{i-1,j} \\ M_{i,j-1} \end{cases} & if \quad i,j > 0 \end{cases} \tag{2a}$$

$$D(T_j, Q_i) = |T_j - Q_i| \tag{2b}$$

In Figure 1 an example of the matrix M for the DTW algorithm is presented. The blue squares represent the target, the green cells represent the query, dark grey cells are the initial values and light grey cells are the computed values. The arrows show the data dependencies. In this example, the values of the orange cells are required to compute the yellow cell, this stencil pattern is the same for all the computed elements (light grey cells).

			T_0	T_1	T_2	T_3	T_4	T_5	...	T_m
			2	3	4	5	6	7	...	3
		0	Inf	Inf	Inf	Inf	Inf	Inf	...	Inf
Q_0	3	Inf	1	1	2	4	7	11	...	...
Q_1	4	Inf	3	2	1	2	4	7	...	...
Q_2	5	Inf	6	4	2	1	2	?	...	...
...	...	...	...	...	...	...	...	...	...	...
Q_n	7	Inf	...	...	...	...	...	...	...	...

Figure 1: DTW matrix example

In the DTW algorithm, the goal often is to know the similarity of the signals without taking into account sample shifts. That implies that we look for the score (or cost of aligning), which is the minimum of the values on the last row. With that consideration, traceback is not required, and therefore, the partial results (besides the last row) only need to be temporarily stored while they are required for computation.

3.2 Graphical Processing Units

GPUs are massively parallel devices with high computation throughput and memory bandwidth, in contrast to CPUs

which are fast sequential devices. That makes GPUs ideal for dealing with highly parallel applications. Its maximum potential, though, is obtained when the arithmetic intensity of an algorithm (ratio between compute operations and amount of data fetched from memory) is higher than the machine's balance point. That is due to the computational throughput being much larger than the memory throughput.

In our study, we consider one CPU and one GPU. That implies that our program's sequential (or modestly parallel) parts should run on the CPU and the parallel parts on the GPU. Ideally, computation on both sides must be overlapped. Note that offloading part of the computation on the GPU implies sending data from the CPU main memory to the GPU memory, through a bus (usually PCIe) with limited bandwidth. That process generates a latency that should be hidden.

Besides its many advantages, doing efficient computation in GPUs is complex as several requirements have to be fulfilled to leverage its full potential.

3.2.1 GPU programming model

The NVIDIA GPUs implement a single instruction multiple threads (SIMT) paradigm, which extends the single instruction multiple data (SIMD) execution paradigm to multithreading, with multiple threads executing the same instruction.

GPUs contain multiple stream multiprocessors (SMs) composed of SIMD cores. Each of those SMs schedules and executes warps (groups of 32 threads that execute the same instruction), which is the minimum scheduling unit. That can generate a performance problem when different threads on the same warp want to execute different instructions (e.g., this can happen when an if statement is present), as threads in a warp are sequentialized. This problem is called thread divergence.

Regarding the memory hierarchy, GPUs have several memory levels. As usual, each level is smaller and faster than its predecessor. We can categorize them into main memory, L2 cache, and L1/shared cache. In the case of main memory and L2 cache, they are shared among all the SMs. Each SM has its own on-chip L1/shared cache. The shared cache occupies the same physical space as L1 but is software managed by the programmer. Additionally, each thread has its registers.

3.2.2 The Hopper architecture

The recently introduced H100 GPU is part of the 9th generation GPU architecture, codenamed Hopper, and contains 144 SMs, has a 60 MB L2 cache, and 80 GB of main memory with up to 3 TB/s of bandwidth. Each SM contains 128 FP32 CUDA Cores, four Tensor Cores, and 256 KB of L1/Shared memory.

3.2.3 CUDA

CUDA is the environment provided by NVIDIA that allows programmers to use an extension of C/C++ or some other languages to write GPU code. Through that language extension, CUDA provides abstractions for the hierarchy of thread groups, shared memories, and barrier synchronization.

CUDA allows the programmer to define kernels, which are C++ functions that execute in parallel in the GPU by a determined number of threads. When defining a kernel, two parameters are required: the number of blocks and threads per block. A block is a group of threads scheduled in the same SM, and they can be grouped in a grid (collection of blocks). Additionally, the Hopper architecture provides thread block clusters, allowing the programmer to group blocks so they are co-scheduled in a GPU processing cluster. This hierarchy allows the programmer to control the level of parallelism. CUDA also allows the programmer to use synchronization primitives to control the program flow and provides different functions so threads within a warp can share data directly from registers.

3.2.4 DPX

DPX is an extension of the instruction set architecture (ISA), introduced by NVIDIA to accelerate dynamic programming algorithms on GPUs. The DPX instructions are exposed to the programmer as low-level intrinsic functions for version 12 of the CUDA compiling toolset. They perform 2-operand and 3-operand maximum and minimum operations as well as addition and maximum fused into a single instruction, in all cases, with optional clamping to zero (*relu* variant). In addition, *16x2* variants of the instructions operate in a vectorial manner, using two 16-bit packed values into a single 32-bit operand, which means that they do the same operation in parallel. Instead of an operation using a 32-bit value two 16-bit operations are performed in a single step. In all cases, the operands can be signed or unsigned integers. Specific hardware support for those functions is provided only for GPU architectures with a compute capability equal to or greater than 9.0 (Hopper). If there is no hardware support available, the compiler software emulates them.

3.3 GPU Performance Engineering

3.3.1 Roofline models

Arithmetic intensity, which is the number of operations performed by an algorithm per each byte it reads or writes from memory, is an insightful metric used to explain performance. The duality between the memory bandwidth of the GPU and the arithmetic intensity of the algorithm determines the theoretical performance bottleneck of our program, which can be memory-bound (memory bandwidth limits performance, with idle compute units since data is not provided fast enough), compute-bound (maximum use of computation resources is used) or latency-bound (the lack of parallelism avoids the GPU scheduler to hide latencies of instructions and memory requests). The best case is when a program's execution is compute-bound because computation throughput is not wasted. Compute-bound performance is easier to achieve with high arithmetic intensity and a large amount of parallelism, both at the thread level and the instruction level. For each byte read or stored by the program, enough operations are required to take advantage of all the resources available at a given moment to lower the total execution time.

Those limitations can be shown in a roofline model [31] as the one in Figure 2, a plot showing the performance

limitations that impose the different memory levels (anti-diagonal lines), the maximum computational throughput (horizontal lines), and the arithmetic intensity of the different algorithms or implementations being analysed (which would be vertical lines). The intersection of the bandwidth (anti-diagonal lines) and the computational throughput (horizontal lines) defines a point that if projected on the X-axis will define two regions. The left one is the memory-bound region, and the right one is the compute-bound region. That means that when representing the arithmetic intensity it will be in one of those two regions and therefore the problem is characterized.

In Figure 2 we can see an example of a roofline model. As shown in the figure, the maximum arithmetical throughput available is 5937 GigaCells per second (yellow horizontal line), and the global memory bandwidth is 1631.23 GB per second (continuous blue anti-diagonal line). The intersection of those lines defines de machine balance point, where we are using the full potential of computing and memory resources. In this case, de memory-bound region is coloured in blue while the compute-bound region is coloured in yellow.

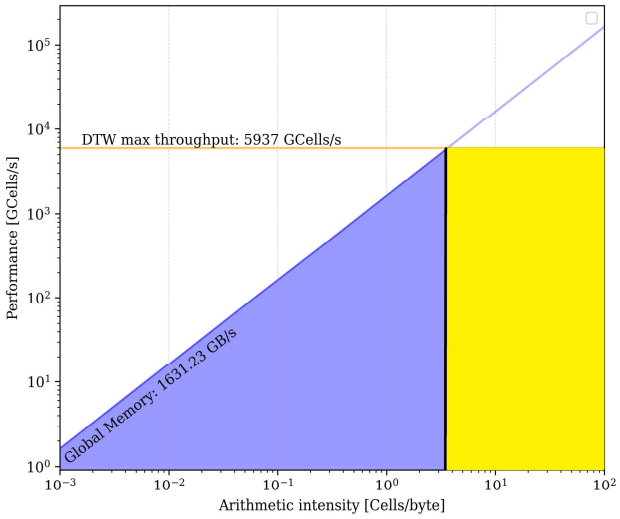

Figure 2: Roofline model example

4 Experimental Methodology

4.1 Experimental set-up

We use a high-end computing node from the supercomputer Mare Nostrum 5 equipped with an Intel Xeon Platinum 8460Y+ (Sapphire Rapids) processor and an Nvidia H100 GPU (Hopper architecture). To compare with a previous GPU generation model, we additionally used a server-grade compute node equipped with an Intel Xeon W-2155 processor (Skylake server) and an NVIDIA RTX 3080 GPU (Ampere architecture). All the GPU applications have been executed on both environments, and CPU applications have been run in the Mare Nostrum 5 node.

Regarding the input datasets, we select 8 representative datasets for our use cases (4 for the NW and 4 for the DTW). Each dataset contains sequences of a fixed length of 100, 250, 1000, or 10000 elements. Elements are bases for NW and positive integer values for DTW.

4.2 Metrics

The theoretical evaluation of the suitability of DPXs to aid in dynamic programming computation is performed through the use of roofline models. For that purpose, we need to know (1) the memory bandwidth (both in global and shared memory), (2) the maximum computational throughput achieved when performing the operations in registers (as it is the memory with the highest bandwidth and lowest latency; i.e., the best case possible), and (3) the arithmetic intensity of the kernels we want to evaluate. Also, to evaluate the dynamic programming programs, we compare the GPU and CPU execution time using different input datasets. Additionally, we compare the throughput they achieved to their theoretical maximum.

4.2.1 DPX instructions

The roofline model can use different units for the computing throughput. Traditionally, Floating Point Operations per Second (FLOPS) are used in applications with intense floating point arithmetic usage (like AI). In our case, we want to explore how the new DPX instructions affect the computational throughput of the GPU. We chose our units to be Tera DPX per second (TDPX/s); this way, the difference between using a real hardware-implemented DPX with a software-emulated one is shown. This study aims to evaluate DPXs and their suitability for DP algorithms. It does not aim to select the best DPX for a specific algorithm, so no comparisons are made between different DPXs.

To obtain the throughput we used microbenchmarking, a technique that uses a very minimal benchmark to get insightful information about a machine's architecture. The microbenchmark is performed with a program that measures the execution time of each specific DPX function for a determined number of calls. To do this, we create a loop that continuously executes a certain DPX instruction. For this approach to work it is important to ensure that the overhead included by the loop does not affect our measured times. In our case, after analysing the generated assembly code, we conclude that the overhead was approximately 9 instructions per iteration. Hence, we decided to introduce loop unrolling with an unroll factor of 100 as higher values do not significantly improve performance. Additionally, it is important to work at the register level to ensure that we are not memory-limited. We also need to verify that the compiler generates the necessary instructions and does not omit them when optimizing. To know the memory bandwidth at the different levels of the memory hierarchy, we used the work published in [29].

4.2.2 Dynamic Programming kernels

To get the maximum arithmetical throughput of the GPU when solving DP algorithms, we consider cells as our smallest units. A cell is the inner part of the DP algorithm's loop, which involves the computation required to obtain the result of one cell of the DP matrix. In the DP algorithms, additional operations need to be performed besides the DPX. Using the cells as a metric allows us to consider the theoretical maximum throughput that can be achieved in each case and simplifies the analysis of DP kernels' performance.

5 Experimental Results

5.1 Static Analysis of DPX

As a first approach to the DPX instructions, a static analysis of the code has been performed. The results can be shown in Table 1. The first column indicates the operation that the intrinsic function executes. The second column shows the operands' size (in bits). The third column indicates whether it clamps the result to zero (*relu* variant) or not. The fourth, sixth, and eighth columns show the number of SASS (GPU machine code) instructions generated for architectures Ampere, which software emulates the instruction, and Hopper, which can use the software-emulated version or the native hardware support. It is important to mention that a reduction in the number of instructions may or may not result in a reduction of the time needed to perform them as different instructions may have different latencies which means they require a different amount of time to be performed.

In Table 1 we can appreciate that the *16x2* DPXs (Which perform 2 16-bit operations at a time) are the ones that improve more in the new architecture. In the case of those variants, if the software emulation takes place, bitwise operations need to be performed. That fact results in a higher instruction count for the emulated version. In the H100 native version, the fact that the *16x2* variants result in the same number of instructions that their equivalent 32-bit version leads us to think that if the size of the problem can fit in a 16-bit representation, the computational throughput (considering alignments/second) can potentially double. Also, that implies that the memory footprint of an alignment is reduced.

In some cases, the number of instructions differs for Ampere and Hopper architectures when generating emulated code, this is because CUDA code is compiled in 2 steps. The first step generates PTX code which depends on the compute capability and the second one depends on the physical GPU microarchitecture. When the first step occurs, and the emulation takes place, the code is translated to PTX. After that, the second step translates the PTX code to SASS, in this step the compiler may use DPXs' SASS instructions (if the architecture can use them) if it considers it is a better option.

Additionally, during the analysis it has been proven that in some cases (specifically the max/min with two operators) the compiler optimizes the code if it detects some operations can be compressed in a maximum with 3 operands. Some of the cases convert 2 function calls to 1 SASS instruction, 3 function calls to 2 SASS instructions and 4 function calls to 3 SASS instructions.

It is important to mention that in the previous static analysis using previous versions of CUDA 12, the use of the DPX functions did not result in the generation of the same SASS instructions. Specifically, the three-operand SASS instructions were not used, resulting in the use of two two-operand maximum instructions and, therefore, a higher count. Therefore, it is highly recommended to use CUDA version 12.3.1 or greater instead of previous versions of CUDA 12 if the hardware supports the new SASS instructions.

5.2 Microbenchmarks

The results obtained throughout the microbenchmarks are also presented in Table 1. It needs to be considered that the results obtained in the H100 for the emulated version of *__vimax_s32_relu* and *__vimax3_s32_relu* are not comparable and therefore discarded, that is due to the compiler optimizations which lead to a misleading result. In those cases, and due to the nature of the operation (signed maximum with clamping to 0), the compiler optimized the code, avoiding part of the computation.

A remarkable characteristic is that when considering the native support for DPX in H100, all the DPX obtain a throughput of around 16.4 TeraDPX/s. That fact means that algorithms that can use 16-bit operands (and therefore *16x2* DPXs) will be able to perform alignment computation around 2 times faster than their 32-bit equivalent, and at between 6 to 10 times faster than software emulating their functionality.

As an additional consideration, the relation between the number of emulated and native instructions for the H100 coincides approximately with the throughput relation in the cases of *max/min* instructions. On the other hand, for the fused *addmax/addmin* instructions, the relation of the number of instructions with the throughput differs. Probably this is due to some computation units, which are not used for the native version, working in parallel to perform some of the additional instructions generated in the emulated version.

Operation	Operands	Relu (clamp to zero)	3080		H100			
			Emulated		Emulated		Native	
			Number of intructions	Throughput (TeraDPXs/s)	Number of intructions	Throughput (TeraDPXs/s)	Number of intructions	Throughput (TeraDPXs/s)
max3	s32		2	3.6	1	16.5	1	16.3
min3	s32		2	3.6	1	16.5	1	16.3
max	s32	relu	2	3.6	2	NC	1	16.5
min	s32	relu	2	3.6	2	8.3	1	16.6
max3	s32	relu	3	2.4	2	NC	1	16.3
min3	s32	relu	3	2.4	2	8.3	1	16.3
max3	u32		2	3.6	1	16.5	1	16.3
min3	u32		2	3.6	1	16.5	1	16.3
max3	s16x2		8	0.9	8	2.0	1	16.3
min3	s16x2		8	0.9	8	2.0	1	16.3
max	s16x2	relu	8	0.9	8	2.0	1	16.6
min	s16x2	relu	8	0.9	8	2.0	1	16.6
max3	s16x2	relu	12	0.6	12	1.4	1	16.3
min3	s16x2	relu	12	0.6	12	1.4	1	16.3
max3	u16x2		6	1.2	6	2.6	1	16.3
min3	u16x2		6	1.3	6	2.7	1	16.3
addmax	s32		2	7.0	1	16.5	1	16.3
addmin	s32		2	7.0	1	16.5	1	16.3
addmax	s32	relu	3	3.6	2	8.3	1	16.3
addmin	s32	relu	3	3.6	2	8.3	1	16.3
addmax	u32		2	7.0	1	16.5	1	16.3
addmin	u32		2	7.0	1	16.5	1	16.3
addmax	s16x2		9	1.2	9	2.7	1	16.3
addmin	s16x2		9	1.2	9	2.7	1	16.3
addmax	s16x2	relu	13	0.7	13	1.6	1	16.3
addmin	s16x2	relu	13	0.7	13	1.6	1	16.3
addmax	u16x2		8	1.2	8	2.7	1	16.3
addmin	u16x2		8	1.2	8	2.7	1	16.3

Table 1: DPX FUNCTIONS THROUGHPUT (TERADPXS/S)

5.3 DP algorithms

5.3.1 GPU-CPU performance

This section presents a comparison between a single-threaded CPU execution of the programs that perform the sequence alignment (i.e., baseline) and two GPU executions of those same programs adapted to GPU. For both algorithms, the comparison has been done using different data sets containing sequences of length 100 (DS_100), 250 (DS_250), 1000 (DS_1K), and 10000 (DS_10K). The number of sequences contained in each input dataset has been adjusted so that the amount of computed cells remains constant for each experiment. The best results for each device are shown in Table 2.

Algorithm	CPU	GPU (3080)	GPU (H100)
NW	0.2	5.7	30.6
DTW	0.1	5.4	29.2

Table 2: CPU-GPU THROUGHPUT (GCELLS/S) IN EACH DEVICE FOR NW AND DTW

For both algorithms, the higher performance corresponds to the H100 and the DS_1K. In those cases, the computation corresponding to the alignment is mainly limited by global memory as can be seen in Figure 3.

5.3.2 DPX impact on Dynamic Programming

Figure 3 compares the maximum throughput obtained when performing alignments accessing only registers and a real implementation that performs one alignment per thread on global memory.

For reference, the maximum throughput that can be achieved using *__vimin3_s32* is 16300 GDPXs/s, as shown in Table 1. As the algorithms (NW and DTW) perform one *min* operation per cell. To serve as a baseline, we can consider that a 32-bit DPX equals 1 cell to make the comparison. Note that if we consider the use of *16x2* DPXs, the equivalence would be that one *16x2* DPX equals two cells resulting in double the computational throughput measured in Cells/s. Therefore, theoretically, if we had implemented the algorithms using the *16x2* variant of the DPX we could consider the throughput would approximately double.

Figure 3 shows that both, NW and DTW, implementations are memory-bound. Operations must be performed at the register level to exploit the full computation potential of the H100; otherwise, memory will be the limiting performance factor. Additionally, Figure 3 illustrates the impact of other operations that need to be performed to compute a cell, reducing the maximum achievable throughput.

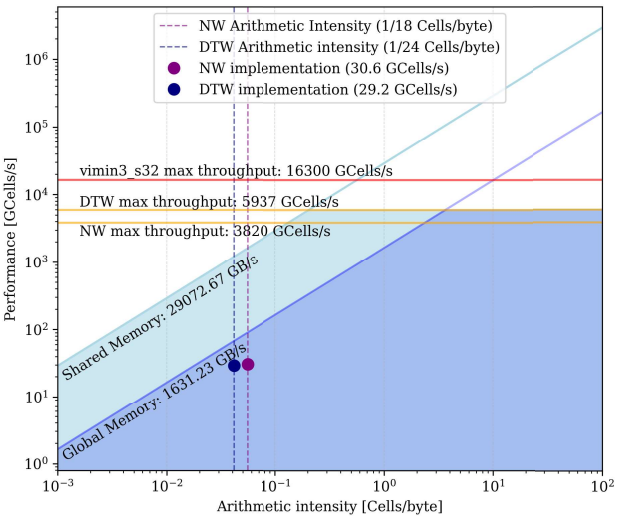

Figure 3: Roofline model considering both algorithms and the equivalence 1 DPX = 1 Cell for *__vimin3_s32*

6 CONCLUSIONS

In this study, we sought to evaluate the newly available DPXs GPU instructions and their impact when accelerating dynamic programming algorithms. Specifically, we measured the improvement introduced by DPXs, compared the performance of an H100 with the use of CPUs and a GPU of the previous architecture, and defined the potential of the H100 for dynamic programming.

In order to do so, we have presented a microbenchmark suite for DPXs in architectures Ampere and Hopper, a dynamic programming GPU benchmark using DPX instructions and an experimental evaluation of programs that implement NW and DTW algorithms in GPU.

As a result, we have demonstrated that the H100 has a huge potential to aid in dynamic programming algorithms computation when using DPXs, specifically the *16x2* variants. But to unlock their maximum potential it is required to use a CUDA version greater or equal to 12.3, an architecture that supports DPXs (i.e. Hopper) and to compile with a compute capability of 9.0 or greater. Nevertheless, for that potential to be exploited, it is necessary to work at the register level and efficiently use the read data exploiting locality.

As future lines regarding dynamic programming, two key obstacles should be addressed: data-sharing, due to the intensive reuse of data, and irregular parallelism, as for each time step, the size of the anti-diagonal that can be computed grows. Both of these problems could be addressed using a tiling strategy. A tiling strategy that divides the matrix into long rectangular tiles would have a region with a fixed diagonal length; therefore, the greater the difference between its height and width, the more stable the parallelism would be. Moreover, considering the data dependencies, a well-dimensioned tile would allow a higher locality. However, squared tiles are not a good approach due to the nature of the data dependencies. Some other approximations, such as the one in [30], could be useful.

REFERENCES

[1] A. Abboud, V. V. Williams, and O. Weimann, "Consequences of faster alignment of sequences," in International Colloquium on Automata, Languages, and Programming. Cham, Switzerland: Springer, 2014, pp. 39–51.

[2] Liu, Y., Wirawan, A. & Schmidt, B. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instruc-

tions. BMC Bioinformatics 14, 117 (2013). https:\\doi.org/10.1186/1471-2105-14-117

[3] Ahmed, N., Lévy, J., Ren, S. et al. GASAL2: a GPU accelerated sequence alignment library for high-throughput NGS data. BMC Bioinformatics 20, 520 (2019). https:\\doi.org/10.1186/s12859-019-3086-9

[4] E. F. d. O. Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro and A. C. M. Melo, "CUDAlign 4.0: Incremental Speculative Traceback for Exact Chromosome-Wide Alignment in GPU Clusters," in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 10, pp. 2838-2850, 1 Oct. 2016, doi: 10.1109/TPDS.2016.2515597.

[5] Awan, M.G., Deslippe, J., Buluc, A. et al. ADEPT: a domain independent sequence alignment strategy for gpu architectures. BMC Bioinformatics 21, 406 (2020). https:\\doi.org/10.1186/s12859-020-03720-1

[6] WFA-GPU: Gap-affine pairwise alignment using GPUs Quim Aguado-Puig, Max Doblas, Christos Matzoros, Antonio Espinosa, Juan Carlos Moure, Santiago Marco-Sola, Miquel Moreto bioRxiv 2022.04.18.488374; doi: https:\\doi.org/10.1101/2022.04.18.488374

[7] Q. Aguado-Puig et al., "Accelerating Edit-Distance Sequence Alignment on GPU Using the Wavefront Algorithm," in IEEE Access, vol. 10, pp. 63782-63796, 2022, doi: 10.1109/ACCESS.2022.3182714.

[8] Barto, Andrew and Bradtke, Steven and Singh, Satinder. (1993). Learning to Act Using Real-Time Dynamic Programming. Artificial Intelligence. 72. 81-138. 10.1016/0004-3702(94)00011-O. https://doi.org/10.1016/0004-3702(94)00011-O.

[9] S. Kumar and E. H. Spafford, "A pattern matching model for misuse intrusion detection," in Proc. Nat. Comput. Secur. Conf., 1994, pp. 11–21.

[10] Wang, Alex. (2010). Don't Follow Me - Spam Detection in Twitter.. 142-151. 10.7312/wang15140-003.

[11] J. Droppo and A. Acero, "Context dependent phonetic string edit distance for automatic speech recognition," in Proc. IEEE Int. Conf. Acoust., Speech Signal Process., Mar. 2010, pp. 4358–4361.

[12] Z. Su, B.-R. Ahn, K.-Y. Eom, M.-K. Kang, J.-P. Kim, and M.-K. Kim, "Plagiarism detection using the levenshtein distance and smith-waterman algorithm," in Proc. 3rd Int. Conf. Innov. Comput. Inf. Control, Jun. 2008, p. 569.

[13] D. Gusfield, "Algorithms on stings, trees, and sequences: Computer science and computational biology," ACM SIGACT News, vol. 28, no. 4, pp. 41–60, Dec. 1997.

[14] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen, "Episode matching," in Proc. Annu. Symp. Combinat. Pattern Matching. Cham, Switzerland: Springer, 1997, pp. 12–27.

[15] T. A. Pirinen and K. Lindén, "State-of-the-art in weighted finite-state spellchecking," in Proc. Int. Conf. Intell. Text Process. Comput. Linguistics. Cham, Switzerland: Springer, 2014, pp. 519–532.

[16] R. C. Gonzalez and M. G. Thomason, Syntactic Pattern Recognition: An Introduction. Reading, MA, USA: Addison-Wesley, 1978.

[17] Z. Ying and T. G. Robertazzi, "Signature searching in a networked collection of files," IEEE Trans. Parallel Distrib. Syst., vol. 25, no. 5, pp. 1339–1348, May 2014.

[18] T. Luczak and W. Szpankowski, "A suboptimal lossy data compression based on approximate pattern matching," IEEE Trans. Inf. Theory, vol. 43, no. 5, pp. 1439–1451, Sep. 1997.

[19] D. Sankoff, "Time warps, string edits, and macromolecules," in The Theory and Practice of Sequence Comparison. Reading, MA, USA: Addison-Wesley, 1983.

[20] S. Jakšić, E. Bartocci, R. Grosu, T. Nguyen, and D. Ničković, "Quantitative monitoring of STL with edit distance," Formal Methods Syst. Design, vol. 53, no. 1, pp. 83–112, Aug. 2018.

[21] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval, vol. 463. New York, NY, USA: ACM Press, 1999.

[22] Yufeng Gu, Arun Subramaniyan, Tim Dunn, Alireza Khadem, Kuan-Yu Chen, Somnath Paul, Md Vasimuddin, Sanchit Misra, Dad Blaauw, Satish Narayanasamy, and Reetuparna Das. 2023. GenDP: A Framework of Dynamic Programming Acceleration for Genome Sequencing Analysis. In Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA'23), June 17–21, 2023, Orlando, FL, USA. ACM, New York, NY, USA, 15 pages. https:\\doi.org/10.1145/3579371.3589060

[23] L. Jiang and F. Zokaee, "EXMA: A Genomics Accelerator for Exact-Matching," 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Korea (South), 2021, pp. 399-411, doi: 10.1109/HPCA51647.2021.00041.

[24] Tae Jun Ham, Dad Bruns-Smith, Brendan Sweeney, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Young H. Oh, Krste Asanoc, Jae W. Lee, and Lisa Wu Wills. Genesis: A Hardware Acceleration Framework for Genomic Data Analysis. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). 254–267.

[25] CUDASW++4.0: Ultra-fast GPU-based Smith-Waterman Protein Sequence Database Search Bertil Schmidt, Felix Kallenborn, Alejandro Chacon, Christian Hundt bioRxiv 2023.10.09.561526; doi: https: doi.org/10.1101/2023.10.09.561526

[26] L. Guo, J. Lau, Z. Ruan, P. Wei and J. Cong, "Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU," 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), San Diego, CA, USA, 2019, pp. 127-135, doi: 10.1109/FCCM.2019.00027.

[27] Sanchit Misra, Tony C Pan, Kanak Mahadik, George Powley, Priya N. Vaidya, Md Vasimuddin, and Srinivas Aluru. 2018. Performance Extraction and Suitability Analysis of Multi- and Many-core Architectures for Next Generation Sequencing Secondary Analysis. In International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus. ACM, New York, NY, USA, 14 pages. https: doi.org/10.1145/3243176.3243197

[28] Albert Gatt and Emiel Krahmer. 2018. Survey of the state of the art in natural language generation: core tasks, applications and evaluation. J. Artif. Int. Res. 61, 1 (January 2018), 65–170.

[29] Konstantinidis, E.; Cotronis, Y., "A quantitative performance evaluation of fast on-chip memories of GPUs", 24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Heraklion, Crete, Greece, pp. 448-455, 2016 doi: 10.1109/PDP.2016.56

[30] Li, Y., Schwiebert, L. Memory-Optimized Wavefront Parallelism on GPUs. Int J Parallel Prog 48, 1008–1031 (2020). https: doi.org/10.1007/s10766-020-00658-y

[31] Williams, Samuel W. (2008). Auto-tuning Performance on Multicore Computers (Ph.D.). University of California at Berkeley.