Manycore high-performance computing in bioinformatics

Jean-Stéphane Varré, Bertil Schmidt, Stéphane Janot, Mathieu Giraud

Manycore high-performance computing in bioinformatics

Jean-Stéphane Varré [1,2]
Bertil Schmidt [3]
Stéphane Janot [1,2]
Mathieu Giraud [1,2]

[1] LIFL, UMR CNRS 8022, Université Lille 1, France
[2] INRIA Lille, France
[3] School of Computer Engineering, Nanyang Technological University, Singapore

March 2010

Published in *Advances in Genomic Sequence Analysis and Pattern Discovery*, edited by L. Elnitski, H. Piontkivska, and L. R. Welch, World Scientific, 2011, ISBN 978-981-4327-72-5.

Abstract

Mining the increasing amount of genomic data requires having very efficient tools. Increasing the efficiency can be obtained with better algorithms, but one could also take advantage of the hardware itself to reduce the application runtimes.

Since a few years, issues with heat dissipation prevent the processors from having higher frequencies. One of the answers to maintain Moore's Law is parallel processing. Grid environments provide tools for effective implementation of coarse grain parallelization. Recently, another kind of hardware has attracted interest: multicore processors.

Graphic processing units (GPUs) are a first step towards massively multicore processors. They allow everyone to have some teraflops of cheap computing power in its personal computer.

The CUDA library (released in 2007) and the new standard OpenCL (specified in 2008) make programming of such devices very convenient. OpenCL is likely to gain a wide industrial support and to become a standard of choice for parallel programming. In all cases, the best speedups are obtained when combining precise algorithmic studies with a knowledge of the computing architectures. This is especially true with the memory hierarchy: the algorithms have to find a good balance between using large (and slow) global memories and some fast (but small) local memories.

In this chapter, we will show how those manycore devices enable more efficient bioinformatics applications. We will first give some insights into architectures and parallelism. Then we will describe recent implementations specifically designed for manycore architectures, including algorithms on sequence alignment and RNA structure prediction. We will conclude with some thoughts about the dissemination of those algorithms and implementations: are they today available on the bookshelf for everyone?

1 Introduction

Most of the algorithms presented in this book require powerful computers to run. These problems are intrinsically complex to solve, and, above all, the amount of input data follows an exponential

curve. With next-generation sequencers (NGS), the data produced every day is growing faster than before. The exponential grows faster, much faster than the growth of computational power.

The challenge for computer scientists and bioinformaticians is to think about new methods and technologies able to deal with this incredible amount of data. A first solution is to design *better algorithms* with more elaborate data structures. However, even if a good algorithm is known, the growth of data still impose to have *better supports of execution*. To increase the computational power, one may use several computers or processors in a grid. Another possibily is to use multicore processors. Now manufacturers manage to put several processors in one: this is what is called dual-core, quad-core, or, more generally, *multicore processors*.

In this chapter, we will see how these recent advances in massively multicore processors, also called *manycore processors*, help to treat the growing flow of bioinformatics data. Manycore processors like Graphics Processing Units (GPUs) offer a huge density of computational units, for an extremely cheap price. They are more and more often directly included in personal computers, making their power available without extra cost. Thus, developments of methods using them have an impact stronger than ever. But the drawback of this technology is that the design of algorithms must take into account the constraints of these architectures. In fact, completely new distributed algorithmics are required, giving new challenges for bioinformaticians. But it is worth the effort: results published in 2008 or 2009 achieve speedups up to 100× compared to serialized one-core algorithms... on commodity GPUs costing less that $500 !

In the following, we briefly describe the evolution of processors, that leads to manycore processors. The evolution of power consumption and heat dissipation led to the so-called *power wall*, that limits the rise of frequencies and commands us to use more and more cores in parallel. In Methods, we present manycore processors and their programming. In Results, we detail algorithms in different fields of bioinformatics, explaining where gains can or cannot be obtained. Finally, we discuss whether these solutions can now be used by everyone, on real applications.

1.1 A small history of processors

A processor, or Central Processing Unit (CPU), is roughly made of three units: the *memory unit* that stores input data, intermediate results and output data, the *instruction unit* that decodes instructions and the *processing unit*, also called arithmetic logic unit (ALU), that actually realizes the computations (Figure 1a) at regular *clock cycles*. The following paragraphs explore two dimensions contributing to the "computational power" of processors: their complexity, driven by the *number of transitors*, and the *frequency* of their clock cycles.

Moore's Law. The number of transistors used to build a chip has dramatically increased, from 2300 for the Intel 4004, released in 1971, to several billions today. Moore's law, formulated in [Moore, 1965] and precised in [Moore, 1975], states that this number doubles every two years. This law has been almost exactly verified since 1960 (Figure 2). More than a natural observation, this is a self-fulfilling prophecy that drives the semiconductor industry.

With more transistors, it is possible to increase the data width (from 4 bits in 1970 to 128 bits or more today), to build more complex operators with more complex instruction sets, and to enhance memory and caches management. This rise of the number of transistors also made possible several improvements in processor design, including instructions pipelines (dividing operators into shorter sections, processing more than one instruction at a time and thus increasing the frequency), superscalar architectures (processing several instructions at a time), out-of-order execution (permuting instructions to better use processor pipelines).

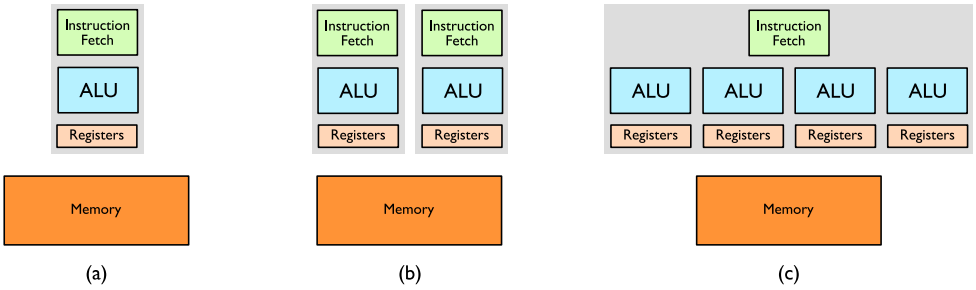

Figure 1: (a) Von Neumann architecture: at each clock cycle, one instruction is fetched from the memory, decoded, and then executed on the ALU. Results are stored back in the registers (very fast, but small, local memory) or in the main memory (slower access). Real processors also contain one or several *caches* (not shown here) between the registers and the main memory. (b) A processor with two cores. Note that each core has its own instruction decoding, and so they can operate independently as long as they do not access the same places in the memory. (c) A SIMD processor. A single instruction decoding unit controls several processing units. Only one flow of instructions is executed, but most of the surface area of the chip is devoted to actual computation.

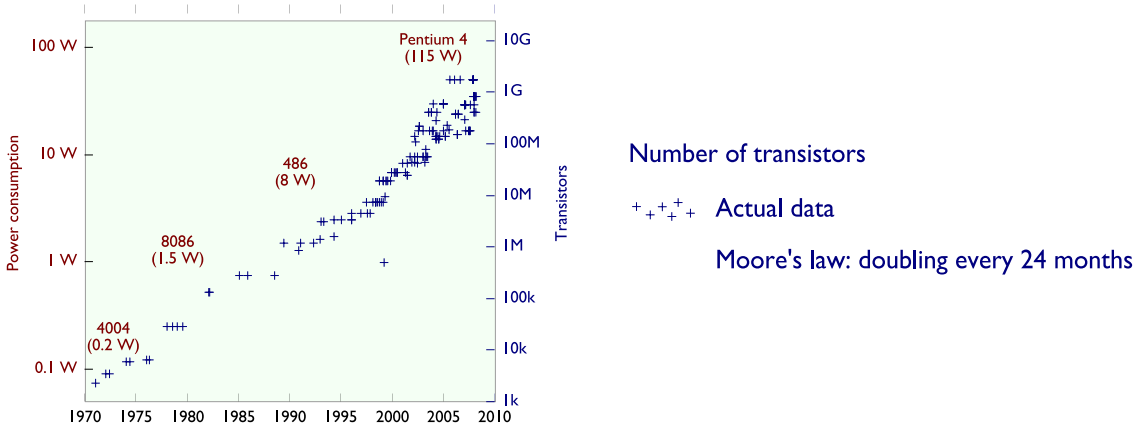

Figure 2: Number of transistors and power consumption of some Intel chips. Scales on Y-axis are logarithmic. Both values have grown at an exponential rate between 1970 and 2005. Data from `http://www.intel.com/`.

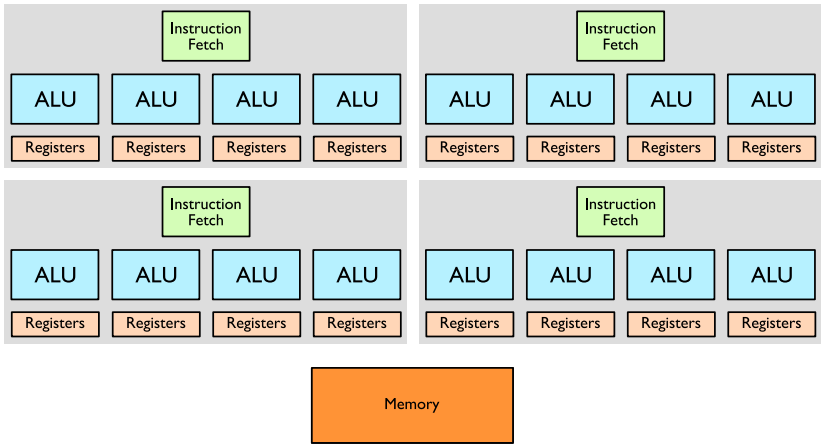

Figure 3: A manycore processor, like current GPUs, combining several cores, each one being a high-density SIMD unit. Again, real processors also contain local memories or caches at different levels between the registers and the main memory.

Frequencies and the "power wall". A common misformulation of the Moore's law is that the "computational power" of the processors doubles every 18 or 24 months. Of course, all techniques cited in the previous paragraph aim to obtain more computational power. Most notably, between 1970 and 2000, the *frequencies* of the processor steadily increased, resulting in major speed gains. Between 1990 and 2000, Intel estimates that the 75× gain in computational power was composed by a 13× gain due to the frequencies and a 6× gain due to processor design improvements [Gelsinger, 2001].

Since a few years, issues with heat dissipation prevent the processors from having higher frequencies. Figure 2 displays some power consumptions of common CPUs. The last Intel mono-core processor, the Pentium 4, achieved a power density of about $100\,\mathrm{W/cm^2}$. If this increase in the frequencies had continued between 2000 and 2010, the thermal density of some processors could today approach the one of a rocket nozzle [Gelsinger, 2001].

Multicore processors. One of the answers to maintain the growth of computational power despite frequencies limits is *parallel processing*. Parallelism is not new: research in distributed algorithms has more than 40 years, with many specialized architectures for high-performance computing. Today, this field of research reaches commodity hardware through grids (several processors or computers) or multicore processors (several cores in a processor).

The idea of *multicore processors* is to have two, four, eight, or even more processors in one (Figure 1b). The high majority of personal computers are provided with this kind of processor at the end of the 2000's. Basically, this can be seen as a "grid on a chip", each core having its own instruction flow. This allows to execute either the same piece of code or different instructions on the different cores, on the same data or on different data.

Data-parallelism and SIMD. To compute the same piece of code on different data, another idea can be implemented. Instead of several processors, we can use a single instruction unit and several processing units (Figure 1c). In this case, the same piece of code is executed at the same time by several processing units. This SIMD paradigm (Single Instruction, Multiple Data) dates from the supercomputers of the 1980's. It becomes really interesting if the same task has to be executed on different data. No surface area is wasted in unnecessary control logic: the processor can be filled

with efficient computing units.

1.2 Towards manycore processors

Multicore processing and SIMD processing can be combined to obtain a higher level of parallelism (Figure 3). This is what is implemented in current *manycore* processors, including Graphical Processing Units (GPUs) and Cell/BE processor. Depending on the design of the processor, emphasis is put either on the multicore or on the SIMD.

GPU processors. Today, GPUs allow everyone to have some teraflops of cheap computing power in its personal computer. High-end GPUs, for no more than $500, embed far more arithmetic units than a CPU of the same price. For example, the NVIDIA GTX 285 has 30 cores, each one being a SIMD unit on 8 x 32 bits at about 600 MHz, whereas the ATI Radeon 4890 has 10 cores, each one being a SIMD unit on 80 x 32 bits, at about 850 MHz.

The GPUs also embed memories with different sizes and capabilities. For example, the NVIDIA GTX 285 has 1 or 2 GB of *global memory*, additional global memories (constant and texture memories), and, for each core, 16 KB of *local memories*. A detailed presentation on some GPUs is available in the review of [Fatahalian and Houston, 2008].

The CPU/GPU convergence. Recent trends blur the line between GPUs and CPUs: CPUs have more and more cores, and cores in GPUs have more and more functions. Some processors are now designed both for graphics and high performance computing.

The Cell Broadband Engine Architecture [Gschwind et al., 2006], or Cell/BE, is the chip included in the Playstation 3. It was designed by IBM and Sony and released in 2006. The Cell includes one CPU-like core, the Power Processor Element (PPE), and 8 GPU-like cores, the Synergistic Processing Elements (SPE), each one being a SIMD unit on 8 x 16 bits.

Larrabee [Seiler et al., 2008] is a Intel graphic processor project. It can be considered as a hybrid between a CPU and a GPU. Basically, each Larrabee core is a simple Pentium core running an extended version of the x86 instruction set, but it also includes a 16 x 32 bits SIMD vector processing unit. The Larrabee was supposed to be released in 2010, but Intel recently announced (in December 2009) that the release is delayed, without giving any new launch date.

Intel argues that the Larrabee processor is more flexible than GPUs. However, next-generation GPUs from NVIDIA (Fermi architecture) and ATI will also have greater flexibility. In 2006, AMD (CPU manufacturer) bought ATI (GPU manufacturer), and their next project, Fusion, is also expected to mix CPU-like and GPU-like cores.

General Purpose computation on GPU. Since the 1980's, graphics hardwares have been used for scientific computations (see for example [Trendall and Stewart, 2000]). The revolution concerning today's "General-Purpose Computation on Graphical Processor Units" (GPGPU) is mainly due to *the increasing number of cores*, making those chips potentially very efficient, and *the availability of programming libraries* for developers non specialized in graphics computing (see in Methods). Some specialized GPUs do not have any video output: such cards are completely dedicated to general computation !

2 Methods

All manycore processors presented in the previous paragraph have a large number of *cores*, each of them being made of a lot of processing sub-units. An algorithm for such hardware thus requires a high level of *parallelism*, that is computations to dispatch to several cores and sub-units. This section details how to program such parallelism, and the next section explains in what kind of bioinformatics applications such parallelism can be used to obtain interesting speedups.

2.1 From GPU tweaks to OpenCL.

General-Purpose Computing on GPU, or GPGPU, was firstly done by tweaking graphics primitives. Operations on color components and pixels have been used to perform, for example, linear algebra computations. Such techniques require expertise in graphics. One of the first popular abstractions proposed for GPU programming is BrookGPU [Buck et al., 2004], developed from 2003 at the Stanford University. It allows to compile and run code in the Brook Stream processing language on NVIDIA and ATI cards.

The CUDA libraries, released by NVIDIA in 2007[1], deeply popularized GPGPU. CUDA is an extension of plain C/C++ (Figure 4), and does not require any knowledge in graphics. Hundreds of applications in various computer science domains are now using CUDA.

```
_global__ void addv(float* a,
                    float* b,
                    float* c)
{
  int i = threadIdx.x ;
  c[i] = a[i] + b[i] ;
}

main()
{
  ...
  addv<<< 1000, 1 >>> (a, b, c);
  ...
}
```

```
__kernel void addv(__global const float* a,
                   __global const float* b,
                   __global float *c)
{
  int i = get_global_gid(0);
  c[i] = a[i] + b[i] ;
}

main()
{
  ...
  kernel = clCreateKernel(..., "addv", ...)
  clEnqueueNDRangeKernel(queue, kernel, ..., 1000, ...);
  ...
}
```

Figure 4: Vector addition in CUDA (left) and OpenCL (right). The `addv` function is the kernel, that is the function executed on each ALU. Each work-item executes this addition on different data, as showed by the instruction modifying the value of `i`. Full codes include initialization of the card, transfer of arrays `a` and `b` from the host to the GPU, and transfer of the resulting array `c` back to the GPU.

The next big step was the specification of a new standard, OpenCL, by a large consortium in December 2008[2]. OpenCL is very close to CUDA, and is likely to gain a wide industrial support. OpenCL offers a unique programming interface to deal with different manycore processors, allowing a same code to be compiled and to run on different chips. But a compiler is needed for each architecture. In 2009, three different implementations, by NVIDIA, AMD/ATI and Apple, were already available, and OpenCL could become in the following years a standard for parallel programming.

[1] `http://www.nvidia.com/cuda`

[2] `http://www.khronos.org/opencl`

In CUDA/OpenCL programs, there are some special functions, called *kernels*, that are executed on the GPU (Figure 4). Due to the architecture of the GPU, such functions are limited: for example, as there is no stack, there are no recursive functions. Ideally, the most intensive computing tasks of an application should be mapped into a kernel. The same kernel is then executed many times by different *work-items*, working on different data sets. The *host program* calls the kernel, but also ensures that data is transferred to and back from the GPU.

2.2 Programming SIMD work-items.

The CUDA/OpenCL kernels on Figure 4 contain an addition instruction, `c[i] = a[i] + b[i]`. As all work-items execute this same instruction, this model is exactly the SIMD paradigm (Figure 1c). This is *implicit SIMD*: the compiler generates code that simultaneously controls all sub-units.

Similar SIMD instructions are available in the common CPUs, but on a much smaller data width. For example, the original SSE instruction set, published in 1999, allows to consider 4 "packed" 32-bits floats in one 128-bit machine word. The SSE assembly instruction `ADDPS` adds two float vectors with one single operation. This instruction can also be compiled from a C "intrinsic" instruction `_mm_add_sd`. Using SSE assembly or intrinsic operations is *explicit SIMD*. Even if it provides more control to the developer, it is not as useful as CUDA or OpenCL for the programmer.

```
for (i=0; i<n; i++)
  {
    if (d[i] > 53)
      c[i]=a[i]+b[i];
    else
      c[i]=d[i];
  }
```

```
for (i=0; i<n; i++)
  {
    unsigned int mask=(d[i] > 53) ? 0xffffffff:0;
    c[i] = (a[i]+b[i]) &  mask)    // then
         | (d[i]       & ~mask);   // else
  }
```

Figure 5: Simulating branching in SIMD. The branching on the left can be simulated by the code on the right. Both branches are executed, and the mask finally chooses the good value for each item.

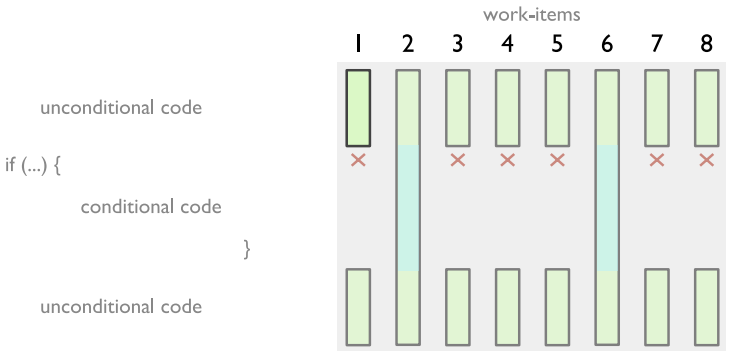

Figure 6: Hardware simulation on branches in a SIMD unit with 8 sub-units. Here only work-items 2 and 6 execute the conditional code: all other work-items are stalled, reducing the overall performance.

2.3 Branches and divergence.

In the SIMD model, at one clock cycle, every arithmetic unit executes the very same instruction, allowing efficient operations on large vectors. But this is not always wanted. For example, depending on a condition in a loop, one could want to execute different instructions (Figure 5, on the left).

Such branches can be *software simulated* in SIMD extensions, by actually running both branches, and finally selecting with a mask the correct value (Figure 5, on the right). Although half of the computations are "wasted" in those branches, a good overall speedup can still be obtained. On some hardware, it is possible to temporarily disable some processing sub-units into a SIMD group. This leads to *hardware simulated* branches: at a given time, only some sub-units can be working in the SIMD processor [Lorie and Strong, 1984].

In CUDA/OpenCL, the developer directly writes a code similar to the left of Figure 5, and the compiler produces code suitable for the GPU, including software or hardware simulation. When, within a same work-group, all work-items do not pass the same branch, a *divergence* occurs and some work-items are stalled (Figure 6) [Coon and Lindholm, 2008, Collange et al., 2009]. Of course, a code with a lot of divergences between work-items results in very poor performance. But, knowing this limitation, it is more simple to write such instructions in CUDA/OpenCL than to explicitly design SIMD masks.

2.4 Work-groups.

Current GPUs are not purely SIMD, because work-items are gathered into independent *work-groups*. There is some global memory available for their communication. This *logical* organization implied by CUDA/OpenCL has different *physical* implementations in actual hardware. For example, the exact relation between the work-groups depends on the efficiency of the global memory and its caching mechanisms. On the current GPU architectures, it is advised to design truly independent work-groups, whereas the Larrabee, with global cache coherence, will allow a more traditional programming with independent threads.

3 Results

Recent parallelizations on GPUs for sequence analysis problems achieve speedups up to 100× compared to a serialized one-core version. This section reviews recent results in different bioinformatics applications.

The first part of this section deals with Smith-Waterman sequence alignments. Manycore processors deliver speeds enabling to run exact dynamic programming (DP) computations rather than incomplete heuristics. The second part explains other algorithms manipulating sequence data, including RNA algorithms and algorithms on weight matrices. Those applications use string algorithmics and DP computations similar to those used by sequence alignments, but on other objects or with different dependencies. Finally, the last part presents a few other applications where kernels deal with non-sequence data.

3.1 Smith-Waterman sequence alignments

In this section we describe how the Smith-Waterman (SW) algorithm for scanning of protein sequence databases can be efficiently mapped onto some manycore architectures. Mapping onto any SIMD architecture requires choosing a fine-grained SIMD vectorization. Mapping to the Cell/BE or a GPU architecture further requires coarse-grained distribution on available cores.

Smith-Waterman Algorithm. The Smith-Waterman (SW) algorithm computes the optimal local pairwise alignment [Smith and Waterman, 1981] of two given sequences S_1 and S_2 of length l_1 and l_2 using DP with the following recurrence relations.

$$\begin{cases} E(i,j) = \max\{E(i,j-1) + g_e, H(i,j-1) + g_o\} \\ F(i,j) = \max\{F(i-1,j) + g_e, H(i-1,j) + g_o\} \\ H(i,j) = \max\{0, E(i,j), F(i,j), H(i-1,j-1) + sbt(S_1[i], S_2[j])\} \end{cases}$$

where $sbt()$ is a substitution matrix such as BLOSUM62 [Henikoff and Henikoff, 1992], g_o is the gap opening penalty, and g_e is the gap extension penalty. The above recurrences are computed for $1 \leq i \leq l_1$ and $1 \leq j \leq l_2$ and are initialized as $H(i,0) = H(0,j) = E(i,0) = F(0,j) = 0$ for $0 \leq i \leq l_1$ and $0 \leq j \leq l_2$.

The score of the optimal local pairwise alignment is the maximal score in matrix H (`maxScore`). The actual alignment can be found by a traceback procedure. However, for SW-based protein sequence database scanning, we just need to compute `maxScore` for each query/database sequence pair. Database sequences are then ranked according to their `maxScore` value and the top hits are displayed to the user. Note that the score-only computation can be done in linear space and does not require storing the full DP matrix. The data dependency in the SW DP matrix (Figure 7) implies that all cells in the same minor diagonal can be computed in parallel.

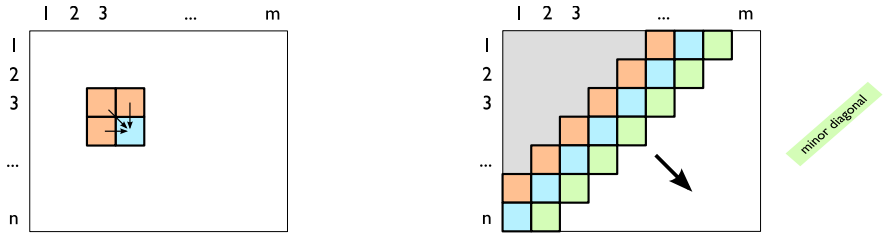

Figure 7: Dependencies in the Smith-Waterman DP matrix. Each cell depends on its left, upper, and upper-left neighbors. All the cells in the same minor diagonal can be computed in parallel.

Mapping onto SIMD registers. Between 1997 and 2007, three approaches have been proposed to vectorize the SW algorithm on CPUs with SIMD instructions sets (Figure 8): (i) *minor diagonal* approach [Wozniak, 1997], (ii) column-based approach with *sequential* memory layout [Rognes and Seeberg, 2000]; and finally (iii) column-based approach with *striped* memory layout [Farrar, 2007].

In order to calculate $H(i,j)$, the value $sbt(S_1[i], S_2[j])$ needs to be added to $H(i-1,j-1)$. The main challenge for any vectorized SW implementation is to avoid performing this table lookup for each element in a SIMD register. Therefore, all three approaches shown in Figure 8 calculate a query profile parallel to the query sequence beforehand. In the minor diagonal approach, the query profile is then used to arrange a SIMD register with all required $sbt()$-values. The advantage of this

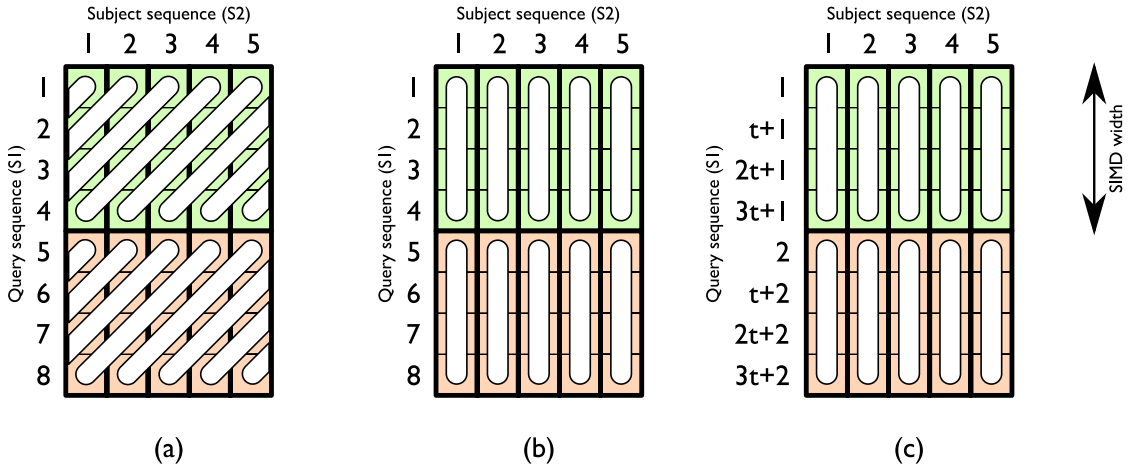

Figure 8: SIMD vectorization approaches: (i) minor diagonal; (ii) column-based with sequential layout; (iii) column-based with striped layout. Each blank box depicts a parallel profile stored into a SIMD register, here with 4 different values.

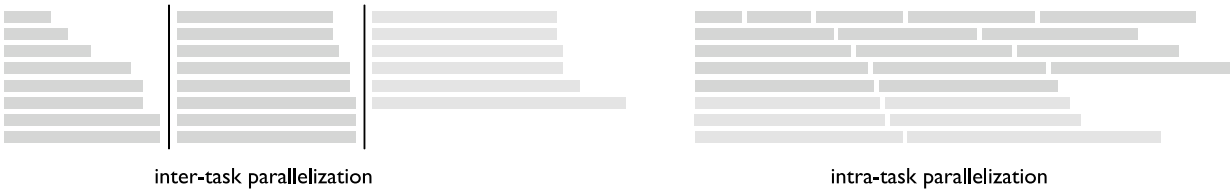

Figure 9: Inter-task vs intra-task parallelization, from [Liu et al., 2009a].

approach is that it does not require any conditional branches in the inner loop. However, it has the disadvantages that this SIMD register has to be updated in each iteration step and that registers are not fully utilized for diagonals at the beginning and at the end. Both column-based approaches have shown higher efficiency due to the simplified dependency relationship and parallel loading of the vector scores from memory. However, a disadvantage introduced by processing in column-based order with sequential memory layout is that a conditional branch is introduced in the inner loop for computing matrix F.

The advantage of the striped layout compared to the sequential layout is that data dependencies between vector registers are moved outside the inner loop. For instance, when calculating vectors for the DP matrices H or F with the sequential layout, the last element in the previous vector has to be moved to the first element in the current vector. When using the striped query layout, this needs to be done just once in the outer loop when processing the next database sequence character.

Implementation on Cell/BE. To fully exploit the capability of the Cell/BE, a parallel SW implementation has to take advantage of the SIMD registers of each SPE. Overall, the column-based approach with striped memory layout is the most efficient of the three approaches and is consequently utilized in publicly available Cell/BE SW implementation such as SWPS3[3] [Szalkowski et al., 2008] and CBESW[4] [Wirawan et al., 2008]. However, it should be mentioned that actual performance depends on the utilized scoring scheme (i.e. substitution matrix and gap penalties).

For coarse-grained parallelization across SPEs, the database can be partitioned into non-overlapping workloads of similar size which are then distributed from the PPE to the different SPEs using multi-threading. Furthermore, due to the limited SPE local store (256KB) a partitioning of the DP matrix for long query sequences is required in SWPS3 [Szalkowski et al., 2008].

Intra-task or inter-task parallelization on GPUs. There are two basic approaches to map SW-based protein sequence databases scanning onto manycore GPUs: (i) inter-task and (ii) intra-task. Considering the SW alignment of a query sequence/database sequence pair as a basic task, the inter-task approach assigns each task to one work-item while the intra-task approach assigns each task to one work-group (Figure 9). Implementation results have shown that inter-task parallelization generally achieves higher performance at the cost of higher memory consumption. Therefore, CUDASW++[5] [Liu et al., 2009a] uses this method for most alignments with the exception of very long database sequences, which use the intra-task approach. To achieve good load balancing within a work-group, all work-items within the same work-group can be assigned database sequences of roughly equal length.

Memory optimization on GPUs. In the CUDA programming model, global memory has to be accessed in a coalesced fashion to achieve high efficiency. Inter-task parallelization uses a coalesced global memory access pattern. A memory slot is allocated to a work-item in a work-group and is indexed top-to-bottom. Therefore, the access to the memory slot uses the same index for all work-items in a work-group. The coalesced global memory arrangement is used for both database sequences and intermediate results of the SW computation.

Fast local memory and registers can be exploited in the SW computation as follows. Instead of computing only one SW cell by each work-item for each global memory access, a whole cell block of size $n \times n$ can be computed. In this case, the computation of n cells in a column (or row) of a cell block only requires one load and one store operation to the global memory instead of n load and n store operations. Since one global access takes hundreds of clock cycles, this method leads to a

[3] `http://www.inf.ethz.ch/personal/sadam/swps3/`

[4] `http://sourceforge.net/projects/cbesw/`

[5] `http://cudasw.sourceforge.net/`

significant performance improvement. However, the size of the cell block is limited by the amount of local memory and registers available; e.g. CUDASW++2.0 uses a size of 8×8 on a GTX280. Furthermore, the gap penalties and query sequence are stored in a constant global memory, while the scoring matrix is loaded to local memory.

		Peak performance	
SSEARCH [Pearson and Lipman, 1988]	CPU (2.0 GHz Xeon)	∼ 0.1 GCUPS	non-SIMD SW implementation
[Farrar, 2007]	CPU (2.0 GHz Xeon)	3.0 GCUPS	striped layout, SSE2 SIMD instructions
CBESW [Wirawan et al., 2008]	Cell/BE	3.6 GCUPS	striped layout, long queries (800)
SWPS3 [Szalkowski et al., 2008]	Cell/BE	9.3 GCUPS	striped layout, long queries
[Farrar, 2009]	Cell/BE	16 GCUPS	striped layout, long queries (32K)
SWCUDA [Manavski and Valle, 2008]	GeForce 8800 GTX	∼ 1.5 GCUPS	inter-task
[Ligowski and Rudnicki, 2009]	GeForce 9800 GX2	7.5 / 14.5 GCUPS	
CUDASW++2.0 [Liu et al., 2009a]	GTX 280/295	16.8 / 28.8 GCUPS	inter-task + intra-task, short/long queries

Table 1: Peak performance of several implementations of Smith-Waterman in GCUPS (Billion Cell Updates Per Second). All Cell/BE and GPU implementations were published in 2008 or 2009. Note that the average performance greatly depends on the input size, as well on the score matrix and gap penalties. The 14.5 and 16.1 GCUPS of Ligowski and CUDASW++ are on dual-GPU cards.

Performance comparison. Table 1 lists some peak performance achieved by recent SW implementations, measured in GCUPS (Billion Cell Updates Per Second). In two years, this active field of research has brought a 10× improvement on almost equivalent hardware. Compared to the non-SIMD plain SW implementation of [Pearson and Lipman, 1988], the best implementations now achieve more than a 100× speedup.

However, peak performances do not measure every aspect of the implementations. For example, in the study of [Liu et al., 2009a], the authors compare the performance of their CUDASW++ on a single-GPU GTX 280 and a dual-GPU GTX 295 versus the performance of SWPS3. SWPS3 achieves a peak performance of around 9.3 GCUPS for longer queries, but is very inefficient for short queries. Due to the inter-task parallelization, the CUDASW++2.0 single-GPU version has a relatively constant performance with an average of 16.3 GCUPS. The dual-GPU version performance is relatively low for short query sequences, but increases up to 28.8 GCUPS for longer ones.

3.2 Algorithms on sequence data

The algorithms discussed in this section alse use kernels working on raw sequence data, but address other bioinformatics problems. The first two algorithms concern DP recurrences, whereas the last one is about weight matrices computations.

RNA folding. Determining the folding of RNAs is important for the study of noncoding RNAs. The secondary structure is a succession of *base pairings*, most often A/T, C/G and G/U. This secondary structure is a basis of the tertiary structure of the RNA. Moreover, comparative genomics has shown that, through evolution, this structure is more conserved than the nucleic sequence.

The Nussinovs algorithm [Nussinov et al., 1978] finds the RNA structure with the maximum number of base pairs. However, complete RNA folding algorithms are typically based on energy minimization, and include energies of stacking regions (or helices), bulge loops, internal loops, hairpin loops and multiple loops, for example as in the full Turner model [Matthews et al., 1999], included in the `mfold/unafold` packages [Zuker, 2003].

In [Rizk and Lavenier, 2009], the authors develop an optimized GPU implementation of `mfold/unafold`, by computing in parallel all cells from the diagonal of the dynamic programming matrix

(Figure 10). At each cell (i, j), if bases i and j can pair, huge computations are run to search for additional RNA structures. However, in the `mfold/unafold` implementation, only 6 over the 16 possible combinations of bases launch these computations. That means that only 6/16 cells of the DP matrix lead to further computations. This breaks the SIMD model of the GPU, as neighbor work-items would probably diverge. To prevent this, the authors pack together these 6/16 cells, and compute all of them in the same time. Finally, they obtain a 17× speedup on a GTX 280 over a one core version on a 2.66 GHz Xeon.

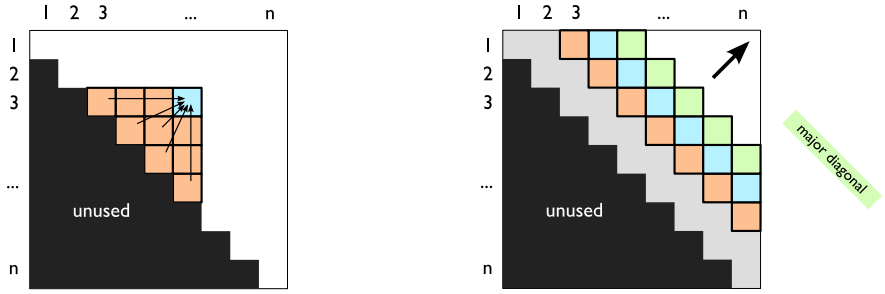

Figure 10: Dependencies in the RNA folding DP matrix, as implemented in [Rizk and Lavenier, 2009]. Each cell depends on its underlying triangle. All cells in the same major diagonal can be computed in parallel.

```
rnafold alg f = axiom struct where
   ...
   struct        = tabulated (sadd <<< base     ~~~ struct |||
                              cadd <<< initstem ~~~ struct |||
                              nil  <<< empty ... h)

   initstem = tabulated (is <<< loc ~~~ closed ~~~ loc ... h)
   closed   = tabulated (
                stack ||| ((hairpin ||| leftB   ||| rightB ||| iloop  ||| multiloop) `with` stackpairing) ... h)

   stack     = (sr   <<< base ~~~ closed ~~~ base)  `with` basepairing ... h
   hairpin   =  hl   <<< base ~~~ base ~~~ (region `with` (minsize 3))      ~~~ base ~~~ base ... h
   leftB     =  bl   <<< base ~~~ base ~~~ region   ~~~ initstem   ~~~ base ~~~ base ... h
   rightB    =  br   <<< base ~~~ base ~~~ initstem ~~~ region     ~~~ base ~~~ base ... h
   iloop     =  il   <<< base ~~~ base ~~~ (region  `with` (maxsize 30)) ~~~ closed ~~~
                                           (region  `with` (maxsize 30)) ~~~ base ~~~ base ... h
   ...
```

Figure 11: Excerpt from the ADP grammar for RNA folding. Starting from the axiom (`struct`), several productions detail how to generate different RNA structure components.

Generic dynamic programming. The Algebraic Dynamic Programming (ADP) framework [Steffen and Giegerich, 2005] is able to encode different DP problems. Starting from an abstract grammar (Figure 11) and an evaluation algebra, the ADP compiler generates the DP dependencies and recurrences, and finally translates them into C (Figure 12). This abstraction is especially useful when the optimization problem includes lots of subcases, resulting in large DP recurrences.

In [Steffen et al., 2009], a CUDA backend of the ADP compiler is proposed[6]. In all ADP programs, all results are combined from results of shorter subsequences. Therefore, the calculation of a table element (i, j) depends only on results that lie in the "underlying triangle" under (i, j), and,

[6] `http://bibiserv.techfak.uni-bielefeld.de/adp/cuda.html`

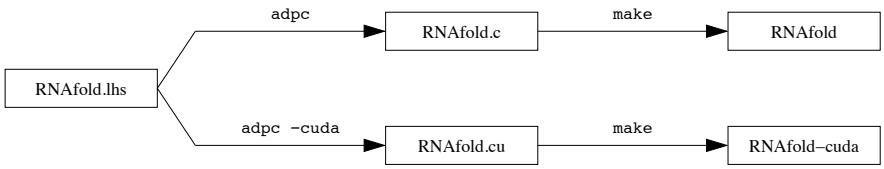

Figure 12: ADP workflow, from [Steffen et al., 2009]. The same abstract grammar and algebra, in `RNAfold.lhs`, is used to produce plain C or CUDA code.

as in the case of RNA folding, such elements can be computed in parallel (Figure 10).

The authors apply this technique on several RNA tools. On pknotsRG, a RNA pseudo-knot detection application [Reeder et al., 2007], they obtain speedups up to 7×. On RNA folding, they achieve a speedup up to 9×, far below the implementation of [Rizk and Lavenier, 2009]. This solution is thus less efficient than a manually crafted implementation but the advantage is definitely the genericity of the approach, even if for now only a few DP problems are encoded like that.

Position Weight Matrices algorithms. Position Weight Matrices (PWMs) model approximates patterns from a set of sequences, as for instance in transcription factor binding sites, splicing sites or protein domains. Given a finite alphabet Σ and a positive integer m, a PWM M is a matrix with $|\Sigma|$ rows and m columns (Figure 13). The PWM associates to each word $u = u_1 u_2 \dots u_p$ the *score* $\sum_{p=1}^{m} M(p, u_p)$. Given a score threshold α, it is possible to compute the probability to achieve a score greater than α: this is the P-value.

There are three usual problems on PWMs: finding positions where a pattern occurs (the *scan*), assessing a P-value for each occurrence, and comparing patterns. In [Giraud and Varré, 2009], a CUDA parallelization achieves speedups of 21× for the brute-force parallelization of the scan and 77× for the P-value computation (on a NVIDIA GTX 280)[7].

The parallelization of the scan is straightforward, distributing positions of the text to different work-items. To reduce memory transfers, work-items in a same work-group process interleaved positions (left of Figure 14).

Computing the P-value is NP-hard [Touzet and Varré, 2007]. The usual algorithm proceeds by recursion [Staden, 1989], computing the score distribution Q_M, where $Q_M(s)$ is the probability to achieve exactly the score s. Then, for a given score s', the P-value is obtained with the equation: $\text{P-value}_M(s') = \sum_{s \geq s'} Q_M(s)$. The distributed algorithm of [Giraud and Varré, 2009] computes an approximation of the score distribution Q_M by splitting the matrix into submatrices. Q is computed for each submatrix and the resulting score distributions are merged (right of Figure 14). Here the brute-force computation of Q for each submatrix takes advantage of the GPU architecture: each work-item evaluates a large set of 4^ℓ words (where ℓ is the length of the submatrix), with no divergence, and no communication nor memory accesses during the main computation. The best speedups are obtained for larger matrices. Unfortunately, for smaller matrices, which are the ones stored into databases, the execution time remains lower using the CPU.

[7] `http://bioinfo.lifl.fr/cudapwm/`

3.3 Other applications

We now briefly review other bioinformatics algorithms that have been parallelized on GPUs. Even if some problems still input sequence data, here the kernels do not directly deal with sequence data.

Indexing structures. As some string algorithms are intrinsically complex, several heuristics have been proposed on CPUs, including the popular BLAST [Altschul et al., 1990] for SW sequence alignments. Heuristics often build indexes when preprocessing a part of the input. Some researches try to put this kind of heuristics on manycore processors.

MUMmer [Kurtz et al., 2004] is an heuristic that uses maximal exact matches (MEM) to seed SW sequence alignments. The MEM detection uses a large suffix tree [Gusfield, 1997]. The paper [Schatz et al., 2007] proposes a CUDA parallelization of MUMmer, splitting the text in 8 Mb pages. [Schatz and Trapnell, 2009] further optimizes this implementation. This latter article contains an exhaustive study of $2^7 = 128$ combinations of design choice for their CUDA implementation[8].

[Shi et al., 2009] elaborates an error correction algorithm for read mapping, based on spectral alignment. Their algorithm uses a specific data structure, a Bloom filter, that is a set of hashing tables with multiple keys. The parallelization is done on the querying of this Bloom filter. Authors report speedups up to 19× compared to an equivalent CPU algorithm.

Phylogeny. [Charalambous et al., 2005] was the first bioinformatics application on GPU. Authors study the RAxML phylogenetics program, and, after profiling, decide to parallelize a particular loop. The speedup (1.2×) was obtained with a Brook kernel on a NVIDIA GeForce FX 5700.

[Suchard and Rambaut, 2009] studies statistical phylogenetics. The goal of such methods is to evaluate the likelihood of a set of sequences given a phylogenetic tree and a model of evolution. The difficulty is in the computation of the likelihood itself. Their implementation uses very precise memory accesses, obtaining a speedup up to 60× on a NVIDIA GTX 280 GPU[9].

Multiple sequence alignment. ClustalW [Larkin et al., 2007] is a multiple sequence alignment program. This progressive alignment technique aims to align the sequences following a guided tree, computed from distance matrices. [Liu et al., 2006] proposes a parallelization of the first phase that builds pairwise alignments, with a method similar to the one presented in the first part of this section.

The study of [Liu et al., 2009b] elaborates a parallelization of the second phase, the Neighbor-Joining algorithm [Saitou and Nei, 1987]. Here the parallelization is on the distance matrix computation, where each cell can be computed independently. The study includes a discussion on the optimal number of work-items and work-groups, as well as on a good memory management taking advantage of the symmetry of the distance matrix. A speedup up to 26× is finally obtained on a NVIDIA GTX 280 card.

Motif finding. CUDA-MEME [Liu et al., 2010] is a CUDA implementation of the popular MEME tool for finding regulatory regions in DNA sequences (so-called motifs). It achieves a speedup of up to 20.5 over the sequential MEME code on a GTX280.

Hidden Markov Models profiles. In [Walters et al., 2009], the authors propose a GPU version of HMMER [Eddy, 1998]. HMMER is a popular software that aims to compute a Hidden Markov Model (HMM) from a set of aligned protein sequences allowing to search for occurrences of similar

[8] `http://www.mummergpu.sourceforge.net`

[9] `http://beagle-lib.googlecode.com/`

proteins in databases. The paralellization focused on the heart of the `hmmmsearch` tool that uses the Viterbi algorithm. A speedup up to 38× is obtained depending on the size of the HMM.

Cell molecules simulation. [Roberts et al., 2009] simulates diffusion of molecules in the cell[10]. The cell topology is modeled as a lattice, and particles follow a random walk. The speedup is not so high (2.4× on a NVIDIA GTX 280), but this study has interesting remarks on the limits of the GPU for this problem.

4 Discussion

The previous section shows that the development of manycore algorithms for bioinformatics has started, taking into account architecture details of such processors. Here we point out what difficulties might be encountered, and discuss how to exploit the processing power of manycore processors in bioinformatics analysis. We also mention some challenges in designing parallel algorithms.

4.1 Challenges in parallel algorithmics

Most of the applications presented in this chapter exhibit *data-parallelism* on sequence data: the same instruction flow is applied to every chunk of data. This is especially the case for DP algorithms (Smith-Waterman sequence comparison, RNA folding, Algebraic Dynamic Programming), but also for other algorithms on sequence data. In all those algorithms, different work-items work on different small sections of the input data. These algorithms are the best ones to parallelize, with almost no divergence between work-items in a SIMD sub-unit. As mentioned in the previous section, researches try to further optimize those algorithms by better memory access patterns or better work-item/work-group balancing.

Another active field of research is to design elaborated data structures intrinsically parallel, as, for example, parallel structures for trees allowing large independent parallel executions. Works on suffix trees [Schatz et al., 2007, Schatz and Trapnell, 2009] are a first step in this direction, but the speedups are not so high for the moment (around 5×). Techniques can be borrowed from graphics community, where researches on ray-tracing also try to conceive good parallel algorithms for trees [Popov et al., 2007].

Of course, new chips with more independent cores could allow more functionalities , but at the price of less dense computational units. Anyway, there is place for research on improved parallel algorithms, using better the resources of manycore architectures.

4.2 Challenges for bioinformatics analysis

Final users of algorithms and methods developed for manycore processors are supposed to be biologists or bioinformaticians who analyze data. Nevertheless, most of these programs are research prototypes rather than end-user applications. In some cases, they need better integration to be, for example, compliant with existing programs, using the same data formats for input and output.

Biomanycores. Biomanycores[11] is intended to be a collection of GPU and manycore bioinformatics tools [Varré et al., 2009]. The goal is both to gather manycore programs and to propose

[10] `http://www.ks.uiuc.edu/Research/gpu/`
[11] `http://www.biomanycores.org`

```
   agatcttaCGTAGTGACGTCtgccatgg
    agatctgGCGGGTGACGTGttgccatgg
   agatcttgGCGGGTGACGTTtctccatgg
                 ...
agatctcggggTATGTTGACGCCatgg
       agatCTTCGTGACGTTctttgccatgg
          aGATCTTGACGTCcgcaggtaccatgg
```

$$M(i,x) = \log_2 \frac{\text{frequency of letter } x \text{ at position } i}{\text{background frequency of letter } x}$$

$$\begin{array}{ll} \text{A} & [\text{-4.85826} \quad \text{-0.06247} \quad \text{-4.85826} \quad \text{-0.46381} \quad \cdots] \\ \text{C} & [\ 0.37462 \quad\ 0.03957 \quad \text{-0.46793} \quad \text{-0.46793} \quad \cdots] \\ \text{G} & [\ 0.03957 \quad \text{-0.18232} \quad\ 0.22107 \quad\ 0.82531 \quad \cdots] \\ \text{T} & [\ 0.22314 \quad\ 0.22314 \quad\ 0.78009 \quad \text{-4.85826} \quad \cdots] \end{array}$$

Figure 13: A Position Weight Matrix (PWM) modeling the CREB1 transcription factor binding site (from the JASPAR database). The coefficients indicate affinities between letters and positions at the binding site: positive if the letter is over-represented in the binding site, else negative. From [Giraud and Varré, 2009].

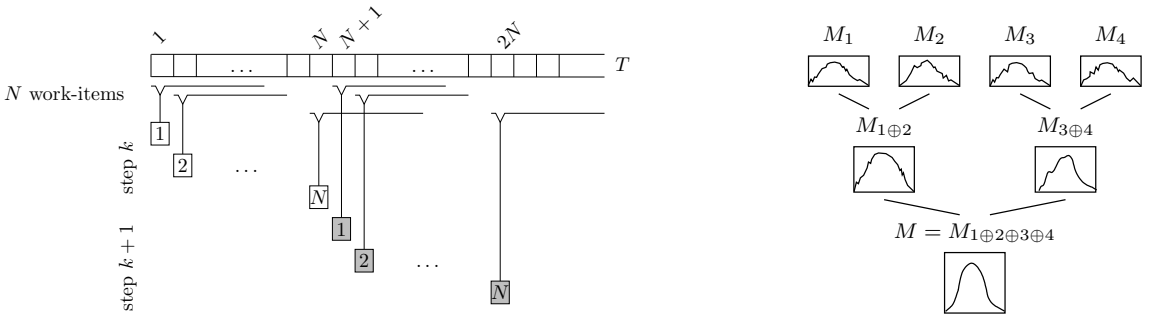

Figure 14: At the left, parallel scan. Positions on the text T are distributed amongst different work-items. At the right, parallel computation of PWM score distribution. The first row is computed in parallel, from brute-force word enumerations. From [Giraud and Varré, 2009].

```
from Bio import SeqIO
from Biomanycores import PadovaSW

bank = SeqIO.parse(open("uniprot.fa"), "fasta")
queries = SeqIO.parse(open("prot.fa"), "fasta")

for query in queries:
    handle = PadovaSW.run(query, bank)
    result = PadovaSW.SWParser().parse()
    print result
```

```
import org.biojavax.bio.seq.RichSequence;
import org.biojava.bio.dp.SimpleWeightMatrix;

import org.biomanycores.bio.pwm.*;
...
{
  RichSequenceIterator it = null;
  BufferedReader in1 = new BufferedReader(new FileReader(args[1]));
  it = RichSequence.IOTools.readFastaDNA(in1, null);
  RichSequence query = it.nextRichSequence();

  // read a weight matrix
  SimpleWeightMatrix pwm =
        PFMParser.PARSER.get(args[2], alph, "ACGT");

  // scan the sequence
  LillePWMScan scanner = new LillePWMScan(launcher);
  List<PWMHit> al = scanner.scan(query,pwm,2500.0);
  ...
}
```

Figure 15: Biomanycores interfaces. At the left, Biopython code for a SW comparison. At the right, BioJava code for a computation with a weight matrix. The interfaces try to use as far as possible the standard objects of existing APIs (such as Biopython's `SeqIO`).

interfaces to Bio* frameworks like BioJava [Holland et al., 2008], BioPerl [Stajich et al., 2002], and Biopython [Cock et al., 2009] (Figure 15).

We mentioned in the previous section several parallel bioinformatics applications already available, especially for sequence alignment, that can provide important speedups with common GPUs. However, those programs are almost not used, or even known, by biologists or bioinformaticians. Biomanycores is based on Bio* frameworks, that are widely used. These frameworks are a collection of tools that allow the user to create specific analysis pipelines for his data. With Biomanycores, the user can keep his pipeline and increase his capacity of analysis by replacing the sequential tool by a parallel one. For example, suppose that a biologist has a large number of sequences to analyse, and wants to format the results. By simply replacing the call to the alignment function by a call to the parallel one, he can have a speedup of around 10x. Concretely, Biomanycores is a repository of open-source parallel bioinformatics code (in Opencl or Cuda) and interfaces to use these programs from Bio* frameworks. It tries to bridge the gap between researches in high-performance-computing and platforms used by bioinformaticians and biologists, by giving access to high-performance prototypes through Bio* frameworks. The project started with three different applications: Smith-Waterman [Manavski and Valle, 2008], pKnotsRG and weight matrices scan (see in section Results). Contributors are welcome to upload their own applications. Of course, the speedup using those APIs is lower than the one obtained by using directly the applications or a manually crafted pipeline, but the advantage is the ease of use inside standard frameworks.

5 Conclusion

Initiated by multicore and graphics processors, the trend of having more cores in a processor will surely keep on during the next years. As explained in the introduction, in the future, the increase of computational power will come from a higher number of processors (either in CPUs and in GPUs) rather than higher frequencies. On the other hand, the amount of biological data produced is growing more and more rapidly, and increases the need for high performance bioinformatics applications. Parallel algorithms could then become the standard, since parallelism is necessary to fully exploit the power of such processors. New tools like CUDA or OpenCL, as detailed in the second section, makes the use of GPUs for general programmning easier, allowing the user to benefit from the computational power of GPUs.

In the third section, we have studied some parallel bioinformatics applications. Parallelism is particularly suitable for bioinformatics algorithms based on dynamic programming, like sequence comparisons. Those applications can achieve good speedups, even on common (and cheap) GPUs. But the use of parallelism is not always straightforward, and more work is needed, for example to design parallel data structures or to find new parallel methods. Bringing parallelism to biologists and bioinformaticians is another challenge, the Biomanycores project is a first step toward this goal.

We believe that the developpement of manycore processors (on commodity hardware) opens a new era in parallel processing, and is an opportunity for high performance computing in bioinformatics.

References

[Altschul et al., 1990] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

[Buck et al., 2004] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for gpus: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23:777–786.

[Charalambous et al., 2005] Charalambous, M., Trancoso, P., and Stamatakis, A. (2005). Initial experiences porting a bioinformatics application to a graphics processor. *Advances in Informatics*, pages 415–425.

[Cock et al., 2009] Cock, P. J. A., Antao, T., Chang, J. T., Toto, T., Titi, T., and XXX, T. (2009). Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*, page btp163.

[Collange et al., 2009] Collange, S., Daumas, M., Defour, D., and Parello, D. (2009). Comparaison d'algorithmes de branchements pour le simulateur de processeur graphique barra. In *RenPar'19 / SympA'2009 / CFSE'7*.

[Coon and Lindholm, 2008] Coon, B. W. and Lindholm, J. E. (2008). System and method for managing divergent threads in a SIMD architecture. US patent 7353369.

[Eddy, 1998] Eddy, S. R. (1998). Profile hidden markov models. *Bioinformatics*, 14:755–763.

[Farrar, 2007] Farrar, M. (2007). Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161.

[Farrar, 2009] Farrar, M. S. (2009). Optimizing Smith-Waterman for the Cell Broadband Engine. `http://farrar.michael.googlepages.com/smith-watermanfortheibmcellbe`.

[Fatahalian and Houston, 2008] Fatahalian, K. and Houston, M. (2008). A Closer Look at GPUs. *Communications of the ACM*, 51(10):50–57.

[Gelsinger, 2001] Gelsinger, P. (2001). Microprocessors for the new millennium: Challenges, opportunities, and new frontiers. In *IEEE International Solid State Circuits Conference (ISSCC 2001)*, pages 22–25.

[Giraud and Varré, 2009] Giraud, M. and Varré, J.-S. (2009). Parallel position weight matrices algorithms. In *International Symposium on Parallel and Distributed Computing (ISPDC 2009)*.

[Gschwind et al., 2006] Gschwind, M., Hofstee, H. P., Flachs, B., Hopkins, M., Watanabe, Y., and Yamazaki, T. (2006). Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24.

[Gusfield, 1997] Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press.

[Henikoff and Henikoff, 1992] Henikoff, J. and Henikoff, S. (1992). Amino Acid Substitution Matrices form Protein Blocks. *Proc. Natl. Acad. Sci. USA*, 89:10915–10919.

[Holland et al., 2008] Holland, R. C. G., Down, T. A., Pocock, M., and al. (2008). BioJava: an open-source framework for bioinformatics. *Bioinformatics*, 24(18):2096–2097.

[Kurtz et al., 2004] Kurtz, S., Phillippy, A., Delcher, A. L., Smoot, M., Shumway, M., Antonescu, C., and Salzberg, S. L. (2004). Versatile and open software for comparing large genomes. *Genome Biology*, 5(R12).

[Larkin et al., 2007] Larkin, M., Blackshields, G., Brown, N., Chenna, R., McGettigan, P., McWilliam, H., Valentin, F., Wallace, I., Wilm, A., Lopez, R., Thompson, J., 3, T. G., and Higgins, D. (2007). Clustal W and Clustal X version 2.0. *Bioinformatics*, 23(21):2947–2948.

[Ligowski and Rudnicki, 2009] Ligowski, L. and Rudnicki, W. (2009). An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*.

[Liu et al., 2006] Liu, W., Schmidt, B., Voss, G., and Müller-Wittig, W. (2006). GPU-ClustalW: Using graphics hardware to accelerate multiple sequence alignment. In *IEEE International Conference on High Performance Computing (HiPC 2006)*, volume 4297 of *Lecture Notes in Computer Science (LNCS)*, pages 363–374.

[Liu et al., 2009a] Liu, Y., Maskell, D., and Schmidt, B. (2009a). CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73.

[Liu et al., 2010] Liu, Y., Schmidt, B., Liu, W., and Maskell, D. (2010). Cuda-meme: Accelerating motif discovery in biological sequences using cuda-enabled graphics processing units. *Pattern Recognition Letters*. In press.

[Liu et al., 2009b] Liu, Y., Schmidt, B., and Maskell, D. (2009b). Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using cuda. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*.

[Lorie and Strong, 1984] Lorie, R. A. and Strong, Jr., H. R. (1984). Method for conditional branch execution in SIMD vector processors. US patent 4435758.

[Manavski and Valle, 2008] Manavski, S. A. and Valle, G. (2008). CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9 Suppl 2:S10.

[Matthews et al., 1999] Matthews, D. H., Sabrina, J., Zuker, M., and Turner, D. H. (1999). Expanded sequence dependence of thermodynamic parameters improves prediction of rna secondary structure. *Journal of Molecular Biology*, 288:911–940.

[Moore, 1965] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

[Moore, 1975] Moore, G. E. (1975). Progress in digital integrated electronics. In *Technical Digest 1975. International Electron Devices Meeting, IEEE*, pages 11–13.

[Nussinov et al., 1978] Nussinov, R., Pieczenik, G., Griggs, J., and Kleitman, D. (1978). Algorithms for loop matchings. *SIAM Journal of Applied Mathematics*, 35:68–82.

[Pearson and Lipman, 1988] Pearson, W. and Lipman, D. (1988). Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci.*, 85:3244–3248.

[Popov et al., 2007] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. (2007). Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424.

[Reeder et al., 2007] Reeder, J., Steffen, P., and Giegerich, R. (2007). pknotsRG: RNA pseudoknot folding including near-optimal structures and sliding windows. *Nucleic Acids Research*, 35(S2):W320–324.

[Rizk and Lavenier, 2009] Rizk, G. and Lavenier, D. (2009). GPU accelerated RNA folding algorithm. In *Using Emerging Parallel Architectures for Computational Science / International Conference on Computational Science (ICCS 2009)*.

[Roberts et al., 2009] Roberts, E., Stone, J., Sepulveda, L., Hwu, W.-M., and Luthey-Schulten, Z. (2009). Long time-scale simulations of in vivo diffusion using GPU hardware. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*.

[Rognes and Seeberg, 2000] Rognes, T. and Seeberg, E. (2000). Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706.

[Saitou and Nei, 1987] Saitou, N. and Nei, M. (1987). The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biololgy and Evolution*, 4(4):406–425.

[Schatz and Trapnell, 2009] Schatz, M. C. and Trapnell, C. (2009). Optimizing data intensive GPGPU computations for DNA sequence alignment. *Parallel Computing*, 35:429:440.

[Schatz et al., 2007] Schatz, M. C., Trapnell, C., Delcher, A. L., and Varshney, A. (2007). High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8:474.

[Seiler et al., 2008] Seiler, L., Carmean, D., Sprangle, E., and *et al.* (14 co-authors) (2008). Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3).

[Shi et al., 2009] Shi, H., Schmidt, B., Liu, W., and Mueller-Wittig, W. (2009). Accelerating error correction in high-throughput short-read dna sequencing data with cuda. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*.

[Smith and Waterman, 1981] Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197.

[Staden, 1989] Staden, R. (1989). Methods for calculating the probabilities of finding patterns in sequences. *Computer Applied Bioscience*, 5(2):89–96.

[Stajich et al., 2002] Stajich, J. E., Block, D., Boulez, K., and al. (2002). The Bioperl toolkit: Perl modules for the life sciences. *Genome Research*, 12(10):1611–1618.

[Steffen and Giegerich, 2005] Steffen, P. and Giegerich, R. (2005). Versatile and declarative dynamic programming using pair algebras. *BMC Bioinformatics*, 6(1).

[Steffen et al., 2009] Steffen, P., Giegerich, R., and Giraud, M. (2009). GPU parallelization of algebraic dynamic programming. In *Parallel Processing and Applied Mathematics / Parallel Biocomputing Conference (PPAM / PBC 09)*.

[Suchard and Rambaut, 2009] Suchard, M. A. and Rambaut, A. (2009). Many-core algorithms for statistical phylogenetics. *Bioinformatics*, 25(11):1370–1376.

[Szalkowski et al., 2008] Szalkowski, A., Ledergerber, C., Krahenbuhl, P., and Dessimoz, C. (2008). SWPS3 - fast multi-threaded vectorized smith-waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes*, 1(1):107.

[Touzet and Varré, 2007] Touzet, H. and Varré, J.-S. (2007). Efficient and accurate p-value computation for position weight matrices. *Algorithms for Molecular Biology*, 2(1).

[Trendall and Stewart, 2000] Trendall, C. and Stewart, A. J. (2000). General calculations using graphics hardware with applications to interactive caustics. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 287–298, London, UK. Springer-Verlag.

[Varré et al., 2009] Varré, J.-S., Janot, S., and Giraud, M. (2009). Biomanycores, a repository of interoperable open-source code for many-cores bioinformatics. In *Bioinformatics Open Source Conference (BOSC)*.

[Walters et al., 2009] Walters, J. P., Balu, V., Kompalli, S., and Chaudhary, V. (2009). Evaluating the use of gpus in liver image segmentation and hmmer database searches. *Parallel and Distributed Processing Symposium, International*, 0:1–12.

[Wirawan et al., 2008] Wirawan, A., Kwoh, C. K., Nim, T. H., and Schmidt, B. (2008). CBESW: Sequence alignment on the Playstation 3. *BMC Bioinformatics*, 9(377).

[Wozniak, 1997] Wozniak, A. (1997). Using video-oriented instructions to speed up sequence comparison. *Computer Applied Bioscience*, 13:145–150.

[Zuker, 2003] Zuker, M. (2003). Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic Acids Research*, 31(13):3406–3415.