Proceedings of the DCICTIA-MICCAI 2012 Workshop

http://www.i3s.unice.fr/~johan/DCICTIA-MICCAI

Data- and Compute-Intensive Clinical and Translational Imaging Applications

Edited by
Johan Montagnat
Preface

The proliferation and increased reliance on high-resolution, multimodality biomedical images/videos present significant challenges for a broad spectrum of clinical practitioners and investigators throughout the clinical and research communities. A growing spectrum of new and improved imaging modalities and new image processing techniques can significantly affect diagnostic and prognostic accuracy and facilitates progress in the areas of biomedical research and discovery. However, the impact of these new technologies in both time-critical clinical applications and high-throughput research pursuits depends, in large part, on the speed and reliability with which the imaging data can be visualized, analyzed and interpreted. Conventional serial computation is grossly inadequate and inefficient for managing these increasing amounts of data and the employment of the new advances in medical imaging is often limited by insufficient compute and storage resources.

High-performance computing (HPC) and distributed computing infrastructures (DCI) ranging from multi-core CPUs and GPU-based processing to parallel machines, Grids and Clouds are effective mechanisms for overcoming such limitations. They allow for significant reduction of computational time, running large experiments campaigns, and speed-up the development time for new algorithms while increasing the availability of new methods for the research community, and supporting large-scale multi-centric collaborations.

The Data- and Compute-Intensive Clinical and Translational Imaging Applications Workshop (DCICTIA-MICCAI’12) was organized with the goals of understanding current trends in HPC/DCI medical imaging research and demonstrating some of the latest developments in the field. Ten papers were submitted to the workshop, out of which 7 were selected for presentation. They report original work related to the application of GPU-based, High-Performance or High-Throughput Computing techniques to medical image analysis, as well as distributed medical databases management. In addition, a keynote talk addressing the challenges and future trends of GPU computing was delivered by Manuel Ujaldón, from the University of Malaga.

The workshop chairs would like to thank all contributors who made the organization of this workshop possible, in particular the Program Committee members, MICCAI organizers, invited speakers and authors.

Nice, October 1st, 2012,

Johan Montagnat
I3S laboratory
CNRS / U. Nice Sophia Antipolis
France

Joel H. Saltz
Emory University
Center for Comprehensive Informatics
USA
Workshop chairs

Johan Montagnat CNRS / I3S, France
Joel H. Saltz Emory University, USA

Program committee

Christian Barillot CNRS / IRISA, France
Umit Catalyurek The Ohio State University, USA
Yann Cointepas CEA, France
Alan Evans MNI / McGill University, Canada
David J. Foran Cancer Institute of New Jersey, USA
Dagmar Krefting University of Applied Sciences Berlin, Germany
Tristan Glatard CNRS / CREATIS, France
Leiguang Gong IBM Watson Research Center, USA
Ron Kikinis Harvard Medical School, USA
Casimir Kulikowski Rutgers University, USA
Tahsin Kurc Emory University, USA
Johan Montagnat CNRS / I3S, France
Toshiharu Nakai National Center for Geriatrics and Gerontology, Japan
Silvia D. Olabarriaga AMC, Netherlands
Joel H. Saltz Emory University, USA
Tony Solomonides CCRI, NorthShore University HealthSystem, USA
George Teodoro Emory University, USA
Manuel Ujaldón University of Malaga, Spain
Lin Yang University of Kentucky, USA
## Table of content

The GPU on High Performance Biomedical Computing  
*Manuel Ujaldón* (invited speaker, University of Malaga)  

Accelerating MI-based B-spline Registration using CUDA Enabled GPUs  
*James Shackleford, Nagarajan Kandasamy and Gregory Sharp*  

A Fast Parallel Implementation of Queue-based Morphological Reconstruction using GPUs  
*George Teodoro, Tony Pan, Tahsin Kurc, Lee Cooper, Jun Kong and Joel Saltz*  

Optimization and Parallelization of the Matched Masked Bone Elimination Method for CTA  
*Henk Marquering, Jeroen Engelbers, Paul Groot, Charles Majoie, Ludo Beenen, Antoine van Kampen and Silvia Olabarriaga*  

Semantic Federation of Distributed Neurodata  
*Gaignard Alban, Johan Montagnat, Catherine Faron Zucker and Olivier Corby*  

Content-based Image Retrieval on Imaged Peripheral Blood Smear Specimens using High Performance Computation  
*Xin Qi, Fuyong Xing, Meghana Ghadge, Ivan Rodero, Moustafa Abdelbaky, Manish Parashar, Evita Sadimin, David Foran and Lin Yang*  

Content-based Parallel Sub-image Retrieval  
*Fuyong Xing, Xin Qi, David Foran, Tahsin Kurc, Joel Saltz and Lin Yang*  

A MapReduce Approach for Ridge Regression in Neuroimaging-Generic Studies  
*Benoit Da Mota, Michael Eickenberg, Soizic Laguitton, Vincent Frouin, Gael Varoquaux, Jean-Baptiste Poline and Bertrand Thirion*
The GPU on High Performance Biomedical Computing

Manuel Ujaldón
CUDA Fellow at Nvidia and
A/Prof at Computer Architecture Department,
University of Malaga (Spain)

Abstract. GPUs (Graphics Processing Units) constitute nowadays a solid alternative for high-performance computing [1]. An increasing number of today’s supercomputers include commodity GPUs to bring us unprecedented levels of performance in terms of GFLOPS/cost. In this paper, we provide an introduction to CUDA programming paradigm which can exploit SIMD parallelism and high memory bandwidth on GPUs. Less demanding alternatives from a biomedical perspective like OpenCL and OpenACC are also briefly described.

Keywords: GPU, High Performance Computing, CUDA, OpenCL, OpenACC.

1 Introduction

We are witnessing a steady transition from multi-core to many-core systems. Many-core GPUs offer excellent performance/cost ratios for scientific applications which fit their architectural idiosyncrasies, and an increasing number of developers are learning to program these processors to take full advantage of their resources via data parallelism.

In this paper, we summarize the ideas required to program GPUs with CUDA (Compute Unified Device Architecture) [2], Nvidia’s programming solution for many-core architectures. We also introduce OpenCL [3], an standardization effort to provide an open standard API, and OpenACC [4], a higher level API using directives for many-core architectures.

1.1 GPU advantages

GPUs gather a number of attractive features as high performance platforms.

At user level, we may highlight two issues: (1) Popularity and low cost, thanks to being manufactured as commodity processors, which means wide availability at an affordable budget, and (2) Easier than ever to be programmed, evolving from the old challenging architecture into a new straightforward platform endowed with an increasing number of libraries, legacy codes, debuggers, profiles, cross-compilers and back-ends.
At hardware level, we remark two other important features: (1) Raw computational power, where GPUs are an order of magnitude ahead (more than a TFLOPS versus 100-150 GFLOPS for a CPU of similar price), and (2) Memory bandwidth, where GDDR5 video memory works up to 6 GHz for a data path 256 bits wide to deliver up to 192 GB/s (a typical CPU operates on a dual-channel motherboard to deliver 128 bits at 1600 MHz using the popular DDR3 memory, which results in 25.6 GB/s of data bandwidth).

1.2 Code programming

Multicore systems can be programmed using task-level parallelism, either relying on the scheduler of the operating system or programming explicitly via pthreads. But when we enter into many-core territory, a different programming method based on data parallelism has to be embraced to enhance scalability. The SIMD paradigm (Single Instruction Multiple Data) re-emerges in this context as a viable solution with a simple idea: We run the same program in all cores, but each thread instantiates on a different data subset for each core to work effectively in parallel. This way, we face a key issue for massive parallelism: data partitioning. Straightforward decompositions are usually derived from 1D, 2D and 3D matrices as input data sets, overall on regular access patterns. Particularly interesting from our biomedical perspective is the area of high-resolution image processing, where massive parallelism benefits from regular data partitioning. On the opposite end, the goal turns more difficult in the presence of irregular elements like data addressed through pointers of arrays accessed through indices unknown at compile-time like another array.

2 GPUs for general-purpose computing

In general, expectations for a particular algorithm to meet growing speedup factors when running on GPUs depend on a number of requirements to be fulfilled. From less to more important, we may cite:

1. Small local data requirements (memory and registers): Programmer skills to decompose computations in tiles.
2. Stream computing, or the ability to avoid recursive algorithms and control and data dependencies.
3. Arithmetic intensity, which is usually coupled with high data reuse.
4. Memory bandwidth, that is, fast movement on high data volumes.
5. Data parallelism. In short, data independency and balanced partitioning.

Fortunately, a generic biomedical application is more likely to satisfy those criteria listed at the end, which makes them good candidates for GPU acceleration. During the rest of the paper, we will illustrate in Section 3 the basic concepts to implement codes on GPUs using CUDA. Section 4 describes some optimizations for threads deployments and memory usage, and finally Section 5 introduces some alternatives that have emerged lately for programming GPUs in high performance computing.
3 CUDA

CUDA [2] is a programming interface and set of supported hardware to enable general purpose computation on Nvidia GPUs. The initiative started in late 2006, and in five years, according to Nvidia and the NSF it has recruited more than 100,000 developers, 1,000,000 compiler downloads, and 15,000,000 developers across very diverse problem domains worldwide, with more than 500 research papers published annually and more than 500 institutions including CUDA in their academic programs.

The CUDA programming interface is ANSI C extended by several keywords and constructs which derive into a set of C language library functions as a specific compiler generates the executable code for the GPU in conjunction with the counterpart version running on the CPU acting as a host.

Since CUDA is particularly designed for generic computing, it can leverage special hardware features not visible to more traditional graphics-based GPU programming, such as small cache memories, explicit massive parallelism and lightweight context switch between threads.

3.1 Hardware platforms

All the latest Nvidia developments on graphics hardware are CUDA-enabled processors: For low-end users and gamers, we have the GeForce series starting from its 8th generation; for high-end users and professionals, the Quadro series; for general-purpose computing, the Tesla boards. Overall, it is estimated to exist more than three hundred fifty million CUDA-enabled GPUs at the end of July, 2012.
3.2 Execution model

CUDA architectures are organized into multiprocessors, each having a number of cores (see Figure 1.a). As technology evolves, future architectures from Nvidia will support the same CUDA executables, but they will be run faster for including more multiprocessors per die, and/or more cores, registers or shared memory per multiprocessor (see Table 1). For example, the GF 100 (Fermi) architecture doubles the number of cores and increases the clock frequency with respect to the previous generation, the GT200. And this one almost doubles the number of multiprocessors and 32-bit register within each multiprocessor compared to the original G80.

The CUDA parallel architecture is a SIMD processor endowed with hundreds of cores which are organized into multiprocessors. Each multiprocessor can run a variable number of threads, and the local resources are divided among them. In any given cycle, each core in a multiprocessor executes the same instruction on different data based on its threadIdx, and communication between multiprocessors is performed through global memory.

The CUDA programming model guides the programmer to expose fine-grained parallelism as required by massively multi-threaded GPUs, while at the same time providing scalability across the broad spectrum of physical parallelism available in the range of GPU devices.

Another key design goal of CUDA is to guarantee a smooth learning curve for most of programmers. To do so, it aims to extend a popular sequential programming language like C/C++ with a minimalist set of abstractions for expressing parallelism. This lets the programmer focus on the important issues of parallelism rather than grappling with the mechanics of an unfamiliar and cumbersome language.

3.3 Memory spaces

The CPU host and the GPU device maintain their own DRAM and address space, referred to as host memory and device memory. The latter can be of three different types. From inner to outer, we have constant memory, texture memory and global memory. They all can be read from or written to by the host and are persistent through the life of the application. Texture memory is the more versatile one, offering different addressing modes as well as data filtering for some specific data formats. Global memory is the actual on-board video memory, usually exceeding 1 GB of capacity and embracing GDDR3/GDDR5 technology. Constant memory has regular size of 64 KB and latency time close to a register set. Texture memory is cached to a few kilobytes. Global and constant memory are not cached at all.

Multiprocessors have on-chip memory that can be of two types: registers and shared memory (see Figure 1.b). Each processor has its own set of local 32-bit read-write registers, whereas a parallel data cache of shared memory is shared by all the processors within the same multiprocessor. The local and global memory spaces are implemented as read-write regions of device memory and were not cached until the Fermi architecture was born.
3.4 Programming Elements

Major elements involved in the conception of a CUDA program which are key for understanding the programming model are shown in Figure 1.b. We summarizes them below.

- A program is decomposed into **blocks** running in parallel. Assembled by the developer, a block is a group of threads that is mapped to a single multiprocessor, where they can share 16 KB or 48 KB of SRAM memory. All the threads in blocks concurrently assigned to a single multiprocessor divide the multiprocessor’s resources equally amongst themselves. The data is also divided amongst all of the threads in SIMD fashion explicitly managed by the developer.

- A **warp** is a collection of 32 threads that can physically run concurrently on all of the multiprocessors. The developer has the freedom to determine the number of threads to be executed, and when there are more threads than the warp size, they are time-shared on the actual hardware resources. This can be advantageous, since time-sharing the ALU resources amongst multiple threads can overlap the memory latencies when fetching ALU operands.

- A **kernel** is a code function compiled to the instruction set of the device, downloaded on it and executed by all of its threads. Threads run on different processors of the multiprocessors sharing the same executable and global address space, though they may not follow exactly the same path of execution, since conditional execution of different operations on each multiprocessor can be achieved based on a unique thread ID. Threads also work independently on different data according to the SIMD model. A kernel is organized into a **grid** as a set of **thread blocks**.

- A **grid** is a collection of all blocks in a single execution, explicitly defined by the application developer, that is assigned to a multiprocessor. The parameters invoking a kernel function call define the sizes and dimensions of the thread blocks in the grid thus generated, and the way hardware groups threads in warps affects performance, so it must be accounted for.

- A **thread block** is a batch of threads executed on a single multiprocessor. They can cooperate together by efficiently sharing data through its shared memory, and synchronize their execution to coordinate memory accesses using the `__syncthreads()` primitive. Synchronization across thread blocks can only be safely accomplished by terminating a kernel. Each thread block has its own thread ID, which is the number of the thread within a 1D, 2D or 3D array of arbitrary size. The use of multidimensional identifiers helps to simplify memory addressing when processing multidimensional data. Threads placed in different blocks from the same grid cannot communicate, and threads belonging to the same block must all share registers and shared memory on a given multiprocessor. This tradeoff between parallelism and thread resources must be wisely solved by the programmer to maximize execution efficiency on a certain architecture given its limitations. These limitations are listed in the lower rows of Table 1 for the past three hardware generations and CUDA compute capabilities.
Table 1. The evolution of CUDA hardware and programming constraints over the past three generations. We take as flagship of each generation what we consider the most popular GPU, together with a representative video memory in graphics cards at that time (around $400 cost at launching date). We highlight in bold font those values which are around an order of magnitude ahead of CPUs of similar cost, namely, peak performance and memory bandwidth.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>GPU architecture for each hardware generation</th>
<th>GT 200</th>
<th>GF 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year released on the marketplace</td>
<td>GFX 80</td>
<td>GTX 280</td>
<td>GTX 480</td>
</tr>
<tr>
<td>Million of transistors within the silicon die</td>
<td>577</td>
<td>602</td>
<td>700</td>
</tr>
<tr>
<td>Number of multiprocessors (SMs)</td>
<td>16</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>SIMD cores on each multiprocessor</td>
<td>8</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Total number of SIMD cores</td>
<td>128</td>
<td>240</td>
<td>480</td>
</tr>
<tr>
<td>Clock frequency for the SIMD cores (GHz)</td>
<td>1.35</td>
<td>1.30</td>
<td>1.40</td>
</tr>
<tr>
<td>Peak performance in GFLOPS (single precision)</td>
<td>none</td>
<td>345</td>
<td>622</td>
</tr>
<tr>
<td>Peak performance in GFLOPS (double precision)</td>
<td>none</td>
<td>1344</td>
<td>672</td>
</tr>
<tr>
<td>SRAM</td>
<td>Shared memory for each multiprocessor</td>
<td>16 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>L1 cache for each multiprocessor</td>
<td>none</td>
<td>none</td>
<td>48/16 KB</td>
</tr>
<tr>
<td>L2 cache for each multiprocessor</td>
<td>none</td>
<td>none</td>
<td>768 KB</td>
</tr>
<tr>
<td>DRAM</td>
<td>Global memory size</td>
<td>512 MB</td>
<td>1 GB</td>
</tr>
<tr>
<td>Frequency for the memory clock (MHz)</td>
<td>2 x 900</td>
<td>2 x 1107</td>
<td>4 x 924</td>
</tr>
<tr>
<td>Bus memory width (bits)</td>
<td>384</td>
<td>512</td>
<td>384</td>
</tr>
<tr>
<td>Memory bandwidth (GB/sc.)</td>
<td>86.4</td>
<td>141.7</td>
<td>177.4</td>
</tr>
<tr>
<td>CUDA compute capabilities</td>
<td>1.0, 1.1</td>
<td>1.2, 1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>Threads / warp (that is, the warp size)</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of thread blocks per multiprocessor</td>
<td>512</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
</tr>
<tr>
<td>Maximum number of threads per multiprocessor</td>
<td>512</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>32-bit registers for each multiprocessor</td>
<td>8192</td>
<td>16384</td>
<td>4096</td>
</tr>
</tbody>
</table>

At the highest level, a program is decomposed into kernels mapped to the hardware by a grid composed of blocks of threads scheduled in warps. No inter-block communication or specific schedule-ordering mechanism for blocks or threads is provided, which guarantees each thread block to run on any multiprocessor, even from different devices, at any time. The number of threads in a thread block is limited to 512 on G80 and GT 200 generations, and to 1024 on GF 100 (Fermi). Therefore, blocks of equal dimension and size that execute the same kernel can be batched together into a grid of thread blocks. This comes at the expense of reduced thread cooperation, but in contrast, this model allows thread blocks of the same kernel grid to run on any multiprocessor, even from different devices, at any time. Again, each block is identified by its block ID, which is the number of the block within a 1D or 2D array of arbitrary size for the sake of a simpler addressing to memory. Kernel threads are extremely lightweight, i.e. creation overhead is negligible and context switching between threads and kernels is essentially free.

3.5 Programming style

CUDA tries to simplify the programming model by hiding thread handling from programmers, i.e. there is no need to write explicit threaded code in the conven-
tional sense. Instead, CUDA includes C/C++ software development tools that allow programmers to combine host code for the CPU with device code for the GPU [7]. To do so, CUDA programming requires a single program written in C/C++ with some extensions [8]:

- Function type qualifiers to specify whether a function executes on the host or on the device and whether it is callable from the host or from the device (\_\_device\_, \_\_global\_, and \_\_host\_).
- Function type qualifiers for functions that execute on the device (\_\_global\_ and \_\_device\_).
- Variable type qualifiers to specify the memory location on the device of a variable (\_\_device\_, \_\_constant\_ and \_\_shared\_).
- Variable type qualifiers for variables that reside on device memory (\_\_device\_, \_\_shared\_ and \_\_constant\_).
- Four built-in variables that specify the grid and block dimensions, the block index within the grid and thread index within the block (gridDim, blockDim, blockIdx and threadIdx), accessible in \_\_global\_ and \_\_device\_ functions.
- An execution configuration construct to specify the dimension of the grid and blocks when launching kernels, declared with the \_\_global\_ directive, from host code (for example, function << gridDim, blockDim, shm_size >> (parameter_list)).

4 Optimizations

Managing those limits introduced by major hardware and software constraints are critical when optimizing applications. Programmers still have a great degree of freedom, though side-effects may occur when deploying strategies to avoid one limit, causing other limits to be hit.

We consider two basic pillars when optimizing a CUDA code: First, organize threads in blocks to maximize parallelism, enhance hardware occupancy and avoid memory banks conflicts. Second, access to shared memory wisely to maximize arithmetic intensity and reduce global memory usage. We now address each of these issues separately.

4.1 Threads deployment

Each multiprocessor contains a number of registers which will be split evenly among all the threads of the blocks assigned to that multiprocessor. Hence, the number of registers needed in the computation will affect the number of threads which can be executed simultaneously, and given the constraints outlined in Table 1, the management of registers becomes important as a limiting factor for the amount of parallelism we can exploit.

The CUDA documentation suggests a block to contain between 128 and 256 threads to maximize execution efficiency. A tool developed by Nvidia, the CUDA
Occupancy Calculator [9], may also be used as guidance to attain this goal. For example, when a kernel instance consumes 16 registers, only 512 threads can be assigned to a single multiprocessor on a G80 GPU. This can be achieved by using one block with 512 threads, two blocks of 256 threads, and so on.

An iterative process is usually followed to achieve the lowest execution time: First, the initial implementation is compiled using the CUDA compiler and a special -cubin flag that outputs the hardware resources (memory and registers) consumed by the kernel on a given hardware generation. Using these values in conjunction with the CUDA Occupancy Calculator, we can analytically determine the number of threads and blocks that are needed to use a multiprocessor with maximum efficiency. When enough efficiency is not achieved, the code has to be revised to reduce the register requirement.

### 4.2 Memory usage

Even though video memory delivers a magnificent bandwidth, it is still a frequent candidate to hold the bottleneck when running the application because of its poor latency (around 400 times slower compared to shared memory) and the high floating point computation performance of the GPU.

Attention must be paid to how the threads access banks of shared memory, since each bank only supports one memory access at a time; simultaneous memory bank accesses are serialized, stalling the rest of the multiprocessor’s running threads until their operands arrive. The use of shared memory is explicit within a thread, which allows the developer to solve bank conflicts wisely. Although such optimization may represent a daunting effort, sometimes can be very rewarding: Execution times may decrease by as much as 10x for vector operations and latency hiding may increase by up to 2.5x [10].

### 5 Programming alternatives

#### 5.1 OpenCL

Almost 1.5 years after CUDA SDK became public, Apple initiated a new standardization effort, and in December 2008 the OpenCL 1.0 specification [11] was released by the Khronos Group, an industry consortium focused on the creation of open standard, royalty-free APIs. The initiative is supported by all major GPU and CPU vendors, including Nvidia, AMD/ATI and Intel.

OpenCL is based on C99, a dialect of C that was adopted as an ANSI standard in May 2000, but it is a low-level standard when compared with other alternatives, primarily targeted at expert developers with few productivity-enhancing abstractions. OpenCL aims to provide a programming environment for those developers to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices using a diverse mix of multicore CPUs, GPUs, Cell-type architectures and other parallel processors such as DSPs. This way, OpenCL should fuel growth and advancement in utilization
of multicore and many-core technologies by addressing two critical problems: Programming challenges and protection of the software investment as hardware technology evolves rapidly.

OpenCL supports both data and task parallelism. Even though the latter does not fit well on GPUs, is applicable to other target architectures like multicores. We may see OpenCL like a superset of CUDA, so programmers may decide to start learning CUDA as an intermediate step towards OpenCL. Or even use Swan [12], a source-to-source translator from CUDA to OpenCL.

OpenCL will enable developers to parallelize an application on a multicore CPU, manycore GPU, or both, amplifying the possibilities for high-performance computation on commodity hardware. Given that each generation of GPU evolution adds flexibility to previous high-throughput GPU designs, software developers in many fields are likely to take interest in the extent to which CPU/GPU architectures and programming systems ultimately converge.

5.2 OpenACC

OpenACC is an open parallel programming standard oriented to average programmers and originally developed by PGI, Cray, CAPS and Nvidia. Initial support by multiple vendors will help accelerate its rate of adoption, providing a stable and portable platform for developer to target.

OpenACC uses directives to enable GPU acceleration as core technology. These directives allow programmers to provide simple hints to the compiler, identifying parallel regions of the code to accelerate without requiring the programmer to modify or adapt the underlying code. The compiler automatically parallelizes loops in your application, which gives you access to the massively parallel processing power of the GPU while making programming straightforward. On platforms that do not support OpenACC directives, the compiler will simply ignore them, ensuring that the code remains portable across many-core GPUs and multi-core CPUs.

5.3 Compilers and tools

A CUDA source code can also be compiled for other target platforms different than Nvidia GPUs (namely, AMD GPUs or x86_64 CPUs) using cross-compilers and compilation environments like the PGI CUDA compiler, Ocelot and MCUDA. And CUDA has also third party wrappers available for Python, Perl, Java, Fortran, Ruby, Lua, Haskell, MatLab and IDL.

It is worth mentioning DirectCompute, Microsoft’s approach for GPU programming. And the latest effort on CUDA interoperability is Libra, released in April’12 by the company GPU Systems as a platform and SDK to uniformly accelerate software applications across major operating systems (Windows, Mac and Linux). This is done via standard programming environments C/C++, Java, C# and Matlab, to execute math operations on top of GPUs and x86_64 CPUs with broad support for compute devices compatible with CUDA, OpenCL and even OpenGL.
6 Concluding remarks

We have described the CUDA programming model and hardware interface as a compelling alternative for assisting non-computer scientists to adapt high-performance biomedical applications to GPUs.

GPUs provide computational power, great scalability and low cost, and may be multiplied on a cluster of GPUs to enhance parallelism and provide even faster responses. Alternatively, we may think of a CPU-GPU hybrid system where an application can be decomposed into two parts to take advantage of the benefits of this bi-processor platform. Recent movements in the hardware industry confirm a tendency towards a stronger CPU-GPU coupling, since not only GPUs have moved closer to CPUs in terms of functionality and programmability, but also CPUs have also acquired functionality similar to that of GPUs. Two good examples are the Fusion project led by AMD which integrates a CPU and GPU on a single chip, and the Knights Corner by Intel to develop another hybrid platform between multicore x86 and a GPU.

The GPU will continue to adapt to the usage patterns of both graphics and general-purpose programmers, with a focus on additional processor cores, number of threads and memory bandwidth available for applications to become a solid alternative for high performance computing. As challenging and computationally demanding areas, biomedical applications constitute of the most exciting fields to take the opportunity offered by CUDA, OpenCL and OpenACC, and those programming models must evolve to include programming heterogeneous many-core systems including both CPUs and GPUs.

References

Accelerating MI-based B-spline Registration using CUDA Enabled GPUs

James A. Shackleford, Nagarajan Kandasamy, and Gregory C. Sharp

1 Department of Radiation Oncology, Massachusetts General Hospital, Boston, MA 02114, USA
2 Electrical and Computer Engineering Department, Drexel University, Philadelphia, PA 19104, USA

Abstract. This paper develops methods for accelerating mutual information based B-spline deformable registration suitable for use on multi-core processors. The registration process is iterative and consists of the following major steps: computing a deformation field from B-spline coefficients, computing the mutual information similarity metric, and computing the change in the mutual information with respect to the B-spline coefficients for the purpose of quasi-Newtonian optimization. We develop parallel algorithms to realize these steps, specifically tailored to multi-core processors including graphics processing units (GPUs). Experimental results using clinical and synthetic data indicate that our GPU-based implementations achieve, on average, a speedup of 21 times with respect to the reference implementation and 7.5 times with respect to a quad-core CPU implementation.

Keywords: B-spline registration, deformable registration, multi-modal registration, mutual information, multi-core, graphics processing units.

1 Introduction

When registering multi-modal images, mutual information (MI) is a useful similarity metric for quantifying the quality of a spatial transform $T$ mapping a reference fixed image $F$ to a destination moving image $M$ by quantifying the amount of information content the two images share in common under the influence of the transform. An optimal registration that accurately maps the voxels of $F$ into $M$ may be obtained by iterative optimization of $T$ such that the shared information content is maximized.

However, using MI as a similarity metric introduces significant additional computational burden to the overall registration process. Therefore, we focus on developing parallel algorithms for B-spline based multi-modal registration using MI that are tailored to multi-core processors including graphics processing units (GPUs). Specifically, our efforts have focused on developing a comprehensive solution that leverages parallelization for the most computationally rigorous aspects of the registration process – namely the generation of the deformation field $T$, the generation of the intensity histograms needed to compute the MI, and
the calculation of the change in the MI with respect to the B-spline coefficients needed for optimization.

Experimental results on clinical and synthetic images are presented with focus on both execution speed and registration quality for the following implementations: a highly optimized single-threaded CPU implementation, a multi-core CPU implementation using OpenMP, and a GPU implementation using the compute unified device architecture (CUDA) interface. For large images, our GPU-based implementations achieve, on average, a speedup of 21 times with respect to the reference implementation and 7.5 times with respect to the OpenMP implementation. These CPU and GPU implementations are freely available online as part of the Plastimatch image registration software and can be downloaded under a BSD-style open source license from www.plastimatch.org.

2 Related Work

Image registration algorithms can be accelerated via the judicial use of parallel processing; in many cases, operations can be performed independently on different portions of the image. For example, the authors of [4, 7] develop parallel deformable registration algorithms on a computer cluster using MI as the similarity metric. Results reported in [7] for B-splines show a speedup of $n/2$ for $n$ processors compared to a sequential implementation; two $512 \times 512 \times 459$ images are registered in 12 minutes using a cluster of 10 computers, each with a 3.4GHz CPU, compared to 50 minutes for a sequential program.

Shams et al. consider the rigid registration of multi-modality images using mutual information and focus on accelerating the histogram generation step on the GPU [6]. The authors apply a concept termed “sort and count” to compute histogram bins in a collision-free manner, thereby eliminating the need for atomic operations. The basic idea uses a parallel version of bitonic sort to order the array of input elements while simultaneously counting the number of occurrences of each unique element in the sorted set. However, the technique is designed for unsigned integers and is therefore incompatible with the partial-volume interpolation required for the effective optimization of the denser parameterization afforded by B-splines. Additionally, the Nvidia SDK provides histogram generation methods specific to indexed images with unsigned integer valued pixels; thus making the method ill-suited for direct application to floating-point image volumes and partial-volume interpolation.

Since the method presented in this paper poses the registration process as an analytic optimization problem, the change in the MI similarity metric with respect to the B-spline parameterization must be computed. This cost-function gradient can be calculated in parallel via the technique developed by Shackleford et al. [5] for accelerating unimodal B-spline registration using the sum of squared differences (SSD). To this end, we provide an extension to this method which allows it to be applied to B-spline registration using MI as the similarity metric.
3 Theory

Given a three dimensional fixed image $F$ with voxel coordinates $\theta = x_1, y_1, z_1$ and voxel intensity $f = F(\theta)$ and moving image $M$ with voxel coordinates $\phi = x_2, y_2, z_2$ and voxel intensity $m = M(\phi)$ representing the same underlying anatomy as $F$ within the image overlap domain $\Omega$, a transform $T(\phi) = \theta + v$ is sought that best registers $M$ within the coordinate system of $F$. Because $F$ and $M$ are obtained using differing imaging modalities, an optimal $T$ will not minimize the error between $f$ and $m$ as in unimodal registration. Instead, we attempt to maximize the amount of information common to $F$ and $M$ using mutual information (MI) as a similarity metric:

$$I = \sum_{f,m} p_j(f, T(m)) \ln \frac{p_j(f, T(m))}{p(f)p(T(m))}$$

which requires that we now view $f$ and $m$ as random variables with associated probability distribution functions $p(f)$ and $p(m)$ and joint probability $p_j(f,m)$. Note that applying $T(\phi)$ to $M$ may result in voxels being displaced outside $\Omega$, thus altering $p(m)$. Additionally, if $T$ maps the point $\theta$ to a location between points in the voxel grid of $M$, some form of interpolation must be employed to obtain $m$, which will modify $p(m)$ in addition to $p_j(f,m)$. These effects are implied within the notation $p(T(m))$ and $p_j(f, T(m))$.

The method of interpolation used to obtain $m$ given $T(\phi)$ can lead to false local maxima emerging in the objective function as $T$ is iteratively optimized. By employing partial volume interpolation [2], it is possible for very small sub-voxel-grid changes in $T$ to manifest in the probability distributions. This is vitally important since these distributions are used to compute both the cost and gradient used to optimize $T$.

Fig. 1(a) depicts a voxel in $F$ at $\theta$ being mapped to a point $\phi$ in $M$ that falls between points in the voxel grid. The interpolation method defines a unity vol-
ume with vertices at center points of the nearest neighbors and then constructs partial volumes that each share the interpolation point \( \phi \) as a common vertex. Once the partial volumes have been computed, they are placed into histogram bins corresponding to the intensity values for the voxels comprising the unity volume vertices, as shown in Fig. 1(b). For example, the fractional weight \( w_1 \) defined by partial volume 1 is placed into the histogram bin associated with neighbor 1, \( w_2 \) in the bin for neighbor 2, etc. This method is used in computing histograms that estimate \( p(T(m)) \) and \( p_j(f, T(b)) \).

As mentioned previously, the gradient used for optimizing \( T \) is also computed using the voxel intensity probability distributions. Furthermore, because we parameterize the coordinate transformation \( T \) in terms of a sparse set of B-spline coefficients \( P \), we may maximize the MI \( I \) through direct optimization of \( P \). An analytic expression for this gradient may be developed by utilizing the chain rule to obtain:

\[
\frac{\partial I}{\partial P} = \frac{\partial I}{\partial v} \times \frac{\partial v}{\partial P} \quad (2)
\]

where the second term is easily obtained by taking the derivative of the B-spline interpolation function used to compute the displacement vectors \( v(\theta) \) that characterize \( T \):

\[
v(\theta) = \sum_{l=1}^{4} \sum_{m=1}^{4} \sum_{n=1}^{4} \beta_l(x)\beta_m(y)\beta_n(z)P_{l,m,n} \quad (3)
\]

where \((l, m, n)\) form the indices of the 64 control points providing local support to the voxel at \((\theta)\), to obtain:

\[
\frac{\partial v}{\partial P} = \sum_{l=1}^{4} \sum_{m=1}^{4} \sum_{n=1}^{4} \beta_l(x)\beta_m(y)\beta_n(z). \quad (4)
\]

which is simply a function of the B-spline basis.

Referring again to the first term of (2), it is important to realize that \( I \) and \( v \) are coupled through the probability distribution \( p_j \) and are therefore directly affected by the 8 neighborhood partial volume interpolation:

\[
\frac{\partial I}{\partial v} = \frac{\partial I}{\partial p_j(f, M(n))} \times \frac{\partial p_j(f, M(n))}{\partial v} \quad (5)
\]

\[
= \sum_{n=1}^{8} \frac{\partial I}{\partial p_j(f, M(n))} \times \frac{\partial w_n}{\partial v}. \quad (6)
\]

where the first term may be found by taking the derivative of (1) with respect to the joint distribution \( p_j \):

\[
\frac{\partial I}{\partial p_j(f, M(n))} = \ln \frac{p_j(f, M(n))}{p(f)p(M(n))} - I. \quad (7)
\]
and the terms $\partial w_n / \partial v$ for $n \in [1, 8]$ are found by taking the derivatives of the eight partial volumes with respect to their common shared vertex.

4 Implementation

To construct the image intensity histograms, $F$ and $M$ are each partitioned into many non-overlapping subregions. Each subregion is assigned to a group of threads called a “thread block,” which ultimately generates a “sub-histogram” for their assigned subregion of the image. The number of voxels in a subregion equals the number of threads within a thread block. The process begins with each thread obtaining the deformation vector $v$ corresponding to its assigned voxel $\theta$ within the subregion of $F$ delegated to the thread block. The thread then computes the eight partial volume weights in $M$ corresponding to $\theta$ in $F$ and subsequently places them into the moving-image histogram bins in parallel as per Fig. 1.

Since all threads within a thread block perform these operations concurrently, we ensure that threads wishing to modify the same histogram bin do not modify the same memory location simultaneously. This is accomplished by representing the histogram in memory as shown in Fig. 2. This data structure, $s_{\text{partitions}}$, resides within the GPU’s shared memory and allows each thread to have its own copy of each histogram bin, which prevents write collisions. Once $s_{\text{partitions}}$ is populated, all threads within the thread block are synchronized and the partitions are merged to generate the sub-histogram data structure $\text{sub hist}$ shown in Fig. 2. After all thread blocks have processed their assigned subregions, the sub-histograms are merged to acquire the final voxel intensity histogram.

While this method efficiently computes $p(f)$ and $p(T(m))$, it cannot be used to generate $p(f, T(m))$ since current GPUs do not have adequate shared memory to maintain individual copies of the joint histogram for each thread in a thread block. We therefore employ atomic operations recently made available in modern GPU models to guarantee mutually exclusive access to histogram bins. The image is once again divided into subregions, each of which is assigned to individual thread blocks to generate the corresponding sub.histograms. However, instead of maintaining a separate partition for each thread within a bin, all threads write to the same bin in shared memory. The GPU instruction $\text{atomicExch}(x, y)$ allows a thread to swap a value $x$ in a shared-memory location with a register value $y$ that is private to the thread, while returning the previous value of $x$. If multiple threads attempt to exchange their private values with the same memory location simultaneously, only one will succeed. This mechanism is used to in combination with a thread-fence to achieve a locking mechanism that emulates atomic floating-point addition on GTX 200 series and newer Nvidia GPUs. Starting with the GTX 300 series, atomic floating-point addition is natively supported; thus providing further increases in execution speed.

Since medical images tend to contain a predominate intensity, that of air for example, the use of atomic operations may lead to severe serialization due to a majority of threads contending to update a single histogram bin. This is avoided
Organization of $s_{\text{partitions}}$:

<table>
<thead>
<tr>
<th>Bin 0</th>
<th>Bin 1</th>
<th>Bin 2</th>
<th>Bin 0</th>
<th>Bin 1</th>
<th>Bin 2</th>
<th>Bin 0</th>
<th>Bin 1</th>
<th>Bin 2</th>
<th>Bin 0</th>
<th>Bin 1</th>
<th>Bin 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread 0</td>
<td>thread 1</td>
<td>thread 2</td>
<td>thread N</td>
<td>thread 0</td>
<td>thread 1</td>
<td>thread 2</td>
<td>thread N</td>
<td>thread 0</td>
<td>thread 1</td>
<td>thread 2</td>
<td>thread N</td>
</tr>
</tbody>
</table>

Organization of $s_{\text{sub_hist}}$:

<table>
<thead>
<tr>
<th>Sub-histogram 0</th>
<th>Sub-histogram 1</th>
<th>Sub-histogram 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bin 0</td>
<td>Bin 1</td>
<td>Bin 2</td>
</tr>
</tbody>
</table>

Fig. 2. Memory organization of data structures used to generate voxel intensity histograms for the fixed and moving images.

by inferring the value for the predominate bin once all other bins have been populated. The inference is simple: all bins must sum to the number of voxels within the overlap domain $\Omega$. Initially, the inferred bin is chosen based on image modality, but as the registration is performed over multiple iterations, the largest bin is tracked and inferred for the next iteration. Experimental results for CT images indicate a noticeable speedup since, on average, 80% of all GPU threads attempt to bin values correlating to air which would otherwise be serialized; thus, inferring effectively mitigates the negative impact on execution speed normally encountered when using atomic memory operations.

Parallel computations for the deformation vectors $v$ and optimization gradient $\partial I/\partial P$ are performed similarly to our unimodal implementation based on the sum of squared differences [5]. These methods carry over easily since computation is largely dominated by the B-spline parameterization and not the employed similarity metric. However, slight extension to the optimization gradient computation is required due to the new form of the $\partial I/\partial v$ term (6). Broadly viewed, the gradient computation consists of three highly parallel stages:

- **Stage 1 is the fetching stage.** In this stage a group of the 64 threads work in unison to fetch a set of 64 contiguous $\partial I/\partial v$ values. Once fetched, Stage 1 ends with each thread computing the local coordinates within the local support region for its $\partial I/\partial v$ value at $\theta$.

- **Stage 2 is the processing stage.** This stage sequentially cycles through each of the 64 possible B-spline piecewise function combinations based on the local coordinates computed in Stage 1. Each function combination is applied to each of the 64 $\partial I/\partial v$ values in parallel; the result is stored into a temporary 64 float wide sum reduction array located in shared memory. Once fully populated, the array is reduced to a single sum, which is then accumulated into a region of shared memory contributions, which is indexed by the piecewise function combination, as in Fig. 3. Stage 2 ends once these operations have been performed for all possible 64 piecewise function combinations. Upon completion, control is returned to Stage 1, which proceeds to begin another cycle by fetching the next set of 64 contiguous $\partial I/\partial v$ values.

- **Stage 3 is the distribution stage.** Once stage 2 has processed all the $\partial I/\partial v$ values within a region of local support, there will be 64 gradient contributions stored within shared memory that need to be distributed to the control points they influence. Because these contributions are indexed
Fig. 3. Memory organization and data flow involved in the parallel computation of the cost-function gradient.

in shared memory using the piecewise basis combination numbers, and since the location of an influenced control-point may be determined from this combination number, distribution becomes simple sorting problem. There are 64 threads and 64 gradient contributions that require distribution – Stage 3 assigns one contribution to each thread and distributes them appropriately based on their combination number. To enable multiple groups of 64 threads to perform this operation in mass-parallel upon different regions of local support, each control point has 64 “slots” in memory for storing the 64 contributions it will receive. Once all 64 slots are populated for all control-point coefficients, a simple sum reduction kernel is employed to add up each control-point coefficient’s 64 slots; thus producing the gradient $\partial I/\partial P$.

5 Results

Here we present runtime and registration quality for common multi-core CPU and GPU architectures; for comparison, a highly optimized single-core CPU implementation is also characterized. The multi-core CPU implementation uses OpenMP [1] to parallelize the processes of histogram generation, cost-function evaluation, and gradient computation. The GPU implementation uses the Nvidia CUDA programming interface to perform both histogram generation and gradient computation on the GPU – cost-function evaluation and quasi-Newtonian optimization remain on the CPU and account for less than 2% of the total compute time. The tests reported here use a machine equipped with a 2.6GHz quad-core Intel i7 920 processor and 12GB of RAM. Performance results are

1 The single-core CPU implementation uses the SSE instruction set when applicable and exhibits a very low miss rate of 0.1% in both L1 and L2 data caches according to the Valgrid profiler [3] for Linux.
Fig. 4. (a) A $512 \times 384 \times 16$ MR volume (red) superimposed upon a $512 \times 512 \times 115$ CT volume (blue) prior to deformable registration. (b) Post registration superimposition produced after 20 L-BFGS-B optimizer iterations using the GPU accelerated implementation. B-spline control-point grid spacing is $100^3$ voxels.

acquired for two different GPUs: the 240-core Tesla C1060 clocked at 1.5GHz with 4GB of onboard memory and the 448-core Tesla C2050 clocked at 1.1GHz with 2.6GB of onboard memory.

Fig. 4 shows the registration results produced by the GPU implementation for a CT-MR pair of thoracic images: a $512 \times 384 \times 16$ voxel MR volume (red) and a $512 \times 512 \times 115$ voxel CT volume (blue). Fig. 4(a) superimposes the images prior to registration and Fig. 4(b) after registration. The CT is the fixed image $F$ and the MR the moving $M$ – the B-spline control-point spacing is $100 \times 100 \times 100$ voxels. The image intensity probability distributions are estimated using histograms of 32 equally-wide bins each for $F$ and $M$; thus, the joint distribution is estimated using 1024 bins. The result shown is produced after 20 L-BFGS-B optimizer iterations.

The effect of input volume resolution on execution time is determined by setting a constant control-point spacing of $15 \times 15 \times 15$ voxels and measuring the execution time of a single optimization iteration for increasingly large synthetically generated input volumes in $10 \times 10 \times 10$ voxel increments. Fig. 5(a) summarizes the results. As expected, the execution time increases linearly with number of voxels. The OpenMP version offers slightly better performance with respect to the reference implementation, with a speedup of 2.5 times. We attribute this to the level of serialization imposed by memory locks used in histogram construction. The GPU achieves a speedup of 21 times with respect to the reference implementation and 8.5 times with respect to the OpenMP implementation. Also, the relatively simple optimization step of inferring the value of the histogram bin expected to incur the most write conflict (in other words, the most number of atomic-exchange operations) significantly improves performance on the Tesla C1060 GPU; for actual CT data, this optimization step speeds up the histogram generation phase by five times. Finally, the \texttt{atomicAdd} instruction available on newer models such as the Tesla C2050 allows histogram bins to be
incremented in a thread-safe manner without the need for atomic-exchanges and thread-fences. Consequently, the Tesla C2050 spends only 60% of its processing time generating histograms whereas the C1060 spends slightly over 70% of its time.

The method used for parallel gradient computation exploits key attributes of the uniform control-point spacing scheme to achieve fast execution times. Each local support region defined by 64 control-points is farmed out to an individual core. If the volume size is constant, increasing control-point spacing results in fewer, yet larger, work units for cores to process. Fig. 5(b) shows the impact of varying the control-point spacing in increments of $5 \times 5 \times 5$ voxels for a $260 \times 260 \times 260$ voxel fixed image. Note that the execution times are relatively unaffected by the control-point spacing for all implementations. The slight sub-linear increase in execution time for the GPU implementations is attributed to the work-unit size becoming adequately large such that the processing time begins to dominate the time required to swap work units in and out to cores.

### 6 Conclusions

We have developed a multi-core accelerated B-spline based deformable registration process for aligning multi-modal images, suitable for use on multi-core processors and GPUs. We developed and implemented parallel algorithms for the major steps of the registration process: generating a deformation field using the B-spline control-point grid, calculating the image histograms needed to compute the mutual information, and calculating the change in the mutual information.
with respect to the B-spline coefficients for the gradient-descent optimizer. We have evaluated the multi-core CPU and GPU implementations in terms of both execution speed and registration quality. Our results indicate that the speedup varies with volume size but is relatively insensitive to the control-point spacing. Our GPU-based implementations achieve, on average, a speedup of 21 times with respect to the reference implementation and 7.5 times with respect to a multi-core CPU implementation using four cores, with identical registration quality. We hope that such improvements in processing speed will result in deformable registration methods being routinely utilized in interventional procedures such as image-guided surgery and image-guided radiotherapy.

The source code for the algorithms presented here have been published online as part of Plastimatch and can be downloaded freely under an open source BSD-style license from www.plastimatch.org.

Acknowledgments

This work was supported in part by NSF ERC Innovation Award EEC-0946463, the Federal share of program income earned by MGH on C06CA059267, and is part of the National Alliance for Medical Image Computing (NAMIC), funded by the National Institutes of Health through the NIH Roadmap for Medical Research, Grant 2-U54-EB005149. Information on the National Centers for Biomedical Computing can be obtained from http://nihroadmap.nih.gov/bioinformatics.

References

A Fast Parallel Implementation of Queue-based Morphological Reconstruction using GPUs

George Teodoro, Tony Pan, Tahsin M. Kurc, Lee Cooper, Jun Kong, and Joel H. Saltz
{gteodor,tony.pan,tkurc,lee.cooper,jun.kong,jhsaltz}@emory.edu
Center for Comprehensive Informatics, Emory University, Atlanta, GA 30322

Abstract. In this paper we develop and experimentally evaluate a novel GPU-based implementation of the morphological reconstruction operation. This operation is commonly used in the segmentation and feature computation steps of image analysis pipelines, and often serves as a component in other image processing operations. Our implementation builds on a fast hybrid CPU algorithm, which employs a queue structure for efficient execution, and is the first GPU-enabled version of the queue-based hybrid algorithm. We evaluate our implementation using state-of-the-art GPU accelerators and images obtained by high resolution microscopy scanners from whole tissue slides. The experimental results show that our GPU version achieves up to 20× speedup as compared to the sequential CPU version. Additionally, our implementation’s performance is superior to the previously published GPU-based morphological reconstruction, which is built on top of slower baseline version of the operation.

1 Introduction

Image data and image analysis have applications in a wide range of domains, including satellite data analyses, astronomy, computer vision, and biomedical research. In biomedical research, for instance, digitized microscopy imaging plays a crucial role in quantitative characterization of biological structures in great detail at cellular and sub-cellular levels. Instruments to capture high resolution images from tissue slides and tissue microarrays have advanced significantly in the past decade. A state-of-the-art slide scanner can capture a 50K×50K pixel image in a few minutes from a tissue sectioned at 3-5 micron thickness, enabling the application of this technology in research and healthcare delivery. Systems equipped with auto-focus mechanisms can process batches of slides with minimal manual intervention and are making large databases of high resolution slides feasible.

In a brain tumor studies, for instance, images created using these devices may contain upto 20 billion pixels (with digitization at 40X objective magnification), or approximately 56 GB in size when represented as 8 bit uncompressed RGB format. To extract meaningful information from these images, they usually are processed through a series of steps such as color normalization, segmentation, feature computation, and classification in order to extract anatomic structures at cellular and sub-cellular levels and classify them [1]. The analysis steps consist
of multiple data and compute intensive operations. The processing of a large image can take tens of hours.

On the computational hardware front, GPGPUs have become a popular implementation platform for science applications. There is a shift towards heterogeneous high performance computing architectures consisting of multi-core CPUs and multiple general-purpose graphics processing units (GPGPUs). Image analysis algorithms and underlying operations, nevertheless, need to take advantage of GPGPUs to fully exploit the processing of such systems.

In this paper, we develop and experimentally evaluate an efficient GPU-based implementation of the morphological reconstruction operation. Morphological reconstruction is commonly used in the segmentation, filtering, and feature computation steps of image analysis pipelines. Morphological reconstruction is essentially an iterative dilation of a marker image, constrained by the mask image, until stability is reached (the details of the operation are described in Section 2). The processing structure of morphological reconstruction is also common in other application domains. The techniques for morphological reconstruction can be extended to such applications as the determination of Euclidean skeletons [8] and skeletons by influence zones [5]. Additionally, Delaunay triangulations [10], Gabriel graphs [2] and relative neighborhood graphs [11] can be derived from these methods, and obtained in arbitrary binary pictures [12].

2 Morphological Reconstruction Algorithms

A number of morphological reconstruction algorithms have been developed and evaluated by Vincent [13]. We refer the reader to his paper for a more formal definition of the operation and the details of the algorithms. In this section we briefly present the basics of morphological reconstruction, including definitions and algorithm versions.

Morphological reconstruction algorithms were developed to both binary and gray scale images. In gray scale images the value of a pixel \( p, I(p) \), comes from a set \( \{0, ..., L-1\} \) of gray levels in a discrete or continuous domain, while in binary images there are only two levels. Figure 1 illustrates the process of gray scale morphological reconstruction in 1-dimension. The marker intensity profile (red) is propagated spatially but is bounded by the mask image’s intensity profile (blue). In a simplified form, the reconstruction \( \rho_I(J) \) of mask \( I \) from marker image \( J \) is done by performing elementary dilations (i.e., dilations of size 1) in \( J \) by a structuring element \( G \). Here, \( G \) is a discrete grid, which provides the neighborhood relationships between pixels. A pixel \( p \) is a neighbor of pixel \( q \) if and only if \( (p, q) \in G \). \( G \) is usually a 4-connected or 8-connected grid. An elementary dilation from a pixel \( p \) corresponds to propagation from \( p \) to its immediate neighbors in \( G \). The basic algorithm carries out elementary dilations successively over the entire image \( J \), updates each pixel in \( J \) with the pixelwise minimum of the dilation’s result and the corresponding pixel in \( I \) (i.e., \( J(p) \leftarrow (\max \{ J(q), q \in N_G(p) \} \} \wedge I(p) \); where \( N_G(p) \) is the neighborhood of a pixel \( p \) on grid \( G \) and \( \wedge \) is the pixelwise minimum operator), and stops when stability is reached, i.e., when no more pixel values are modified.
Vincent has presented several morphological reconstruction algorithms, which are based on this core technique [13]:

**Sequential Reconstruction (SR):** Pixel value propagation in the marker image is computed by alternating raster and anti-raster scans. A raster scan starts from the pixel at \((0, 0)\) and proceeds to the pixel at \((N - 1, M - 1)\) in a row-wise manner. An anti-raster scan starts from the pixel at \((N - 1, M - 1)\) and moves to the pixel at \((0, 0)\) in a row-wise manner. Here, \(N\) and \(M\) are the resolutions of the image in \(x\) and \(y\) dimensions, respectively. In each scan, values from pixels in the upper left or the lower right half neighborhood are propagated to the current pixel in raster or anti-raster fashion, respectively. The raster and anti-raster scans allow for changes in a pixel to be propagated in the current iteration. The SR method iterates until stability is reached, i.e., no more changes in pixels are computed.

**Queue-based Reconstruction (QB):** In this method, a first-in first-out (FIFO) queue is initialized with pixels in the regional maxima. The computation then proceeds by removing a pixel from the queue, scanning the pixel’s neighborhood, and queuing the neighbor pixels that changed. The overall process continues until the queue is empty. The regional maxima needed to initialize the queue requires significant computational cost to generate.

**Fast Hybrid Reconstruction (FH):** This approach incorporates the characteristics of SR and QB algorithms, and is about one order of magnitude faster than the others. It first makes one pass using the raster and anti-raster scans as in SR. After that pass, it continues the computation using a FIFO queue as in QB.

A pseudo-code implementation of FH is presented in Algorithm 1, \(N^+\) and \(N^-\) denote the set of neighbors in \(N_G(p)\) that are reached before and after touching pixel \(p\) during a raster scan.
Algorithm 1 Fast Hybrid gray scale reconstruction

Input
I: mask image
J: marker image, defined on domain \( D_I, J \leq I \).

1: Scan \( D_I \) in raster order.
2: Let \( p \) be the current pixel
3: \( J(p) \leftarrow (\max\{J(q), q \in N_G^+(p) \cup \{p\}\}) \land I(p) \)
4: Scan \( D_I \) in anti-raster order.
5: Let \( p \) be the current pixel
6: \( J(p) \leftarrow (\max\{J(q), q \in N_G^-(p) \cup \{p\}\}) \land I(p) \)
7: if \( \exists q \in N_G^-(p) \mid J(q) < J(p) \) and \( J(q) < I(q) \)
8: \( \text{fifo}_\text{add}(p) \)
9: \{Queue-based propagation step\}
10: while \( \text{fifo}_\text{empty()} = \text{false} \) do
11: \( p \leftarrow \text{fifo}_\text{first()} \)
12: for all \( q \in N_G(p) \) do
13: if \( J(q) < J(p) \) and \( I(q) \neq J(q) \) then
14: \( J(q) \leftarrow \min\{J(p), I(q)\} \)
15: \( \text{fifo}_\text{add}(q) \)

3 Fast Parallel Hybrid Reconstruction Using GPUs

To the best of our knowledge, the only existing GPU implementation of morphological reconstruction is a modified version of SR [4]: SR_GPU. As discussed before, SR is about an order of magnitude slower than FH. Therefore, SR_GPU is built on top of a slow base line algorithm, limiting its gains when compared to the fastest CPU algorithm — FH. This section describes our GPU parallelization of the Fast Hybrid Reconstruction algorithm, referred to in this paper as FH_GPU.

The FH_GPU, as its CPU based counterpart, consists of a raster and anti-raster scan phase that is followed by a queue-based phase. To implement the raster and anti-raster scanning in our algorithm we extended the approach of Karas [4]. We have made several changes to SR_GPU: (1) we have templated our implementation to support binary images as well as gray scale images with integer and floating point data types; (2) we have optimized the implementation and multi-threaded execution for 2-dimensional images — the original implementation was designed for 3-dimensional images. This modification allows us to specialize the Y-direction propagation during the scans; and (3) we have tuned the shared memory utilization to minimize communication between memory hierarchies. For a detailed description of the GPU-based raster scan phase we direct the reader to the paper [4]. The rest of this section discusses the queue-based phase, which is the key aspect to achieve efficiency on execution of morphological reconstruction and is the focus of this work.

3.1 Queue-based Phase Parallelization Strategy

After the raster scan phase, active pixels, those that satisfy the propagation condition to a neighbor pixels, are inserted into a global queue for computation.
The global queue is then equally partitioned into a number of smaller queues. Each of these queues is assigned to a GPU thread block. Each block can carry out computations independently of the other blocks.

Two levels of parallelism can be implemented in the queue-based computation performed by each thread of block: (i) pixel level parallelism which allows pixels queued for computation to be independently processed; and (ii) the neighborhood level parallelism that performs the concurrent evaluation of a pixel neighborhood. The parallelism in both cases, however, is available at the cost of dealing with potential race conditions that may occur when a pixel is updated concurrently.

Algorithm 2 GPU-based queue propagation phase

1: {Split initial queue equally among thread blocks}
2: while queue_empty() = false do
3: while (p = dequeue(...)!) = EMPTY do in parallel
4: for all q ∈ NG(p) do
5: if J(q) < J(p) and I(q) ≠ J(q) then
6: oldval = atomicMax(&J(q), min{J(p), I(q)})
7: if oldval < min{J(p), I(q)} then
8: queue_add(q)
9: if tid = 0 then
10: queue_swap_in_out()
11: _syncthreads()
12: _threadfence_block()

During execution, a value max operation is performed on the value of the current pixel being processed (p) and that of each pixel in the neighborhood (q ∈ NG(p)) — limited by a fixed mask value I(q). To correctly perform this computation, atomicity is required in the maximum and update operations. This can only be achieved at the cost of extra synchronization overheads. The use of atomic CAS operations is sufficient to solve this race condition. The efficiency of atomic operations in GPUs has been significantly improved in the last generation of NVidia GPUs (Fermi) [9] because of the use of cache memory. Atomic operations, however, still are more efficient in cases where threads do not concurrently try to update the same memory address. When multiple threads attempt to update the same memory location, they are serialized in order to ensure correct execution. As a result, the number of operations successfully performed per cycle is reduced [9].

To lessen the impact of the potential serial execution, our GPU parallelization employs the pixel level parallelism only. The neighborhood level parallelism is problematic in this case because the concurrent processing and updating of pixels in a given neighborhood likely affect one another, since the pixels are located in the same region of the image. Unlike traditional graph computing algorithms (e.g., Breadth-First Search [3, 6]), the lack of parallelism in neighborhood processing does not result in load imbalance in the overall execution, because all pixels have the same number of neighbors which is defined by the structuring element G.
The GPU-based implementation of the queue propagation operation is presented in Algorithm 2. After splitting the initial queue, each block of threads enters into a loop in which pixels are dequeued in parallel and processed, and new pixels may be added to the local queue as needed. This process continues until the queue is empty. Within each loop, the pixels queued in last iteration are uniformly divided among the threads in the block and processed in parallel (Lines 3—8 in Algorithm 2). The value of each queued pixel \( p \) is compared to every pixel \( q \) in the neighborhood of pixel \( p \). An atomic maximum operation is performed when a neighbor pixel \( q \) should be updated. The value of pixel \( q \) before the maximum operation is returned \((\text{oldval})\), and used to determine whether the pixel’s value has really been changed (Line 7 in Algorithm 2). This step is necessary because there is a chance that another thread might have changed the value of \( q \) between the time the pixel’s value is read to perform the update test and the time that the atomic operation is performed (Lines 5 and 6 in Algorithm 2). If the maximum operation performed by the current thread has changed the value of pixel \( q \), \( q \) is added to the queue for processing in the next iteration of the loop. Even with this control, it may happen that between the test in line 7 and the addition to the queue, the pixel \( q \) may have been modified again. In this case, \( q \) is added multiple times to the queue, and although it may impact the performance, the correctness of the algorithm is not affected.

After computing pixels from the last iteration, pixels that are added to the queue in the current iteration are made available for the next iteration, and all writes performed by threads in a block are guarantee to be consistent (Lines 9 to 12 in the algorithm). As described in the next section, the choice to process pixels in rounds, instead of making each pixel inserted into the queue immediately accessible, is made in our design to implement a queue with very efficient read performance (dequeue performance) and with low synchronization costs to provide consistency among threads when multiple threads read from and write to the queue.

3.2 Parallel Queue Implementation and Management

The parallel queue is a core data structure employed by FH_GPU. It is used to store information about pixels that are active in the computation and should be processed. An efficient parallel queue for GPUs is a challenging problem [3, 4, 6] due to the sequential and irregular nature of accesses.

A straightforward implementation of a queue (named Naïve here), as presented by Hong et al. [3], could be done by employing an array to store items in sequence and using atomic additions to calculate the position where each item should be inserted. Hong et al. stated that this solution worked well for their use case. The use of atomic operations, however, is inefficient when the queue is heavily employed as in our algorithm. Moreover, a single queue that is shared among all thread blocks introduces additional overheads to guarantee consistency across threads in the entire device.

We have designed a parallel queue that operates independently in a per thread block basis to avoid inter-block communication, and is built with a cascade of storage levels in order to exploit the GPU fast memories for efficient
write and read accesses. The parallel queue (depicted in Figure 2) is constructed in multiple-levels: (i) Per-Thread queues (TQ) which are very small queues private to each thread, residing in the shared memory; (ii) Block Level Queue (BQ) which is also in the shared memory, but is larger than TQ. Write operations to the block level queue are performed in thread warp-basis; and (iii) Global Block Level Queue (GBQ) which is the largest queue and uses the global memory of the GPU to accumulate data stored in BQ when the size of BQ exceeds the shared memory size.

The use of multiple levels of queues improves the scalability of the implementation, since portions of the parallel queue are utilized independently and the fast memories are employed more effectively. In our queue implementation, each thread maintains an independent queue (TQ) that does not require any synchronization to be performed. In this way threads can efficiently store pixels from a given neighborhood in the queue for computation. Whenever this level of queue is full, it is necessary to perform a warp level communication to aggregate the items stored in local queues and write them to BQ. In this phase, a parallel thread warp prefix-sum is performed to compute the total number of items queued in individual TQs. A single shared memory atomic operation is performed at the Block Level Queue to identify the position where the warp of threads should write their data. This position is returned, and the warp of threads write their local queues to BQ.

Whenever a BQ is full, the threads in the corresponding thread block are synchronized. The current size of the queue is used in a single operation to calculate the offset in GBQ from which the BQ should be stored. After this step, all the threads in the thread block work collaboratively to copy in parallel the contents of QB to GBQ. This data transfer operation is able to achieve high throughput, because the memory accesses are coalesced. It should be noted that, in all levels of the parallel queue, an array is used to hold the queue content. While the contents of TQ and BQ are copied to the lower level queues when full, GBQ does not have a lower level queue to which its content can be copied. In the current implementation, the size of GBQ is initialized with a
fixed tunable memory space. If the limit of GBQ is reached during an iteration of the algorithm, excess pixels are dropped and not stored. The GPU method returns a boolean value to CPU to indicate that some pixels have been dropped during the execution of the method kernel. In that case, the algorithm has to be re-executed, but using the output of the previous execution as input, because this output already holds a partial solution of the problem. Hence, recomputing using the output of the previous step as input will result in the same solution as if the execution was carried out in a single step. The operation of reading items from the queue for computation is embarrassingly parallel, as we statically partitioning pixels queued in the beginning of each iteration.

4 Experimental Results

We evaluate the performance of the GPU-enabled fast hybrid morphological reconstruction algorithm (FH_GPU), comparing it to that of existing fastest CPU and GPU implementations under different configurations. Test images used in the experiments were extracted from the set of images collected from a brain tumor research study. Each test image had been scanned at 20× magnification resulting in roughly 50K×50K pixels. The images used in this studies are partitioned into tiles to perform image analyses on a parallel machine. A tile is processed by a single GPU or a CPU core. Hence, the performance numbers presented in this paper are per individual tiles.

We used a system with a contemporary CPU (Intel i7 2.66 GHz) and two NVIDIA GPUs (C2070 and GTX580). Codes used in our evaluation were compiled using “gcc 4.2.1”, “-O3” optimization flag, and NVidia CUDA SDK 4.0. The experiments were repeated 3 times, and, unless stated, the standard deviation was not observed to be higher than 1.3%.

4.1 Results

The evaluation of morphological reconstruction, as the input data size varies, using 4-connected and 8-connected grids is presented in Table 1. The speedups are calculated using the single core CPU version as the baseline.

The results show that both SR_GPU and FH_GPU achieve higher speedups with 8-connected grids. The performance differences are primarily a consequence of the GPU’s ability to accommodate the higher bandwidth demands of morphological reconstruction with 8-connected grid, thanks to the higher random memory access bandwidth of the GPU. For example, when the input tile is 4K×4K, the CPU throughput, measured as a function of the number of pixels visited and compared per second, increases from 67 to 85 million pixels per second as the grid changes from 4-connected to 8-connected. For the same scenario, the FH_GPU has a much better throughput improvement, going from 746 to 1227 million pixels per second. This difference in computing rates highlights that morphological reconstruction has abundant parallelism primarily in memory operations (like other graph algorithms [3, 7]), which limits the performance of the CPU for the 8-connected case. The use of an 8-connected grid is usually
Table 1. Impact of the input tile size on the performance of (i) FH: the fastest CPU version, (ii) SR_GPU: the previous GPU implementation [4], and (iii) FH_GPU: our queue-based GPU version.

<table>
<thead>
<tr>
<th>Input Size</th>
<th>FH_CPU (ms)</th>
<th>SR_GPU (ms)</th>
<th>FH_GPU (ms)</th>
<th>SR_GPU speedup</th>
<th>FH_GPU speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K×4K</td>
<td>1970</td>
<td>1578</td>
<td>1097</td>
<td>1.26</td>
<td>1.97</td>
</tr>
<tr>
<td>8K×8K</td>
<td>8491</td>
<td>5393</td>
<td>1034</td>
<td>646</td>
<td>1.57</td>
</tr>
<tr>
<td>12K×12K</td>
<td>19174</td>
<td>11773</td>
<td>6688</td>
<td>2334</td>
<td>1.62</td>
</tr>
<tr>
<td>16K×16K</td>
<td>34818</td>
<td>19278</td>
<td>10990</td>
<td>3960</td>
<td>1.80</td>
</tr>
</tbody>
</table>

4-connected grid

<table>
<thead>
<tr>
<th>Input Size</th>
<th>FH_CPU (ms)</th>
<th>SR_GPU (ms)</th>
<th>FH_GPU (ms)</th>
<th>SR_GPU speedup</th>
<th>FH_GPU speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K×4K</td>
<td>1371</td>
<td>599</td>
<td>401</td>
<td>169</td>
<td>2.19</td>
</tr>
<tr>
<td>8K×8K</td>
<td>5269</td>
<td>1646</td>
<td>1089</td>
<td>502</td>
<td>3.19</td>
</tr>
<tr>
<td>12K×12K</td>
<td>11767</td>
<td>3156</td>
<td>1952</td>
<td>1023</td>
<td>3.72</td>
</tr>
<tr>
<td>16K×16K</td>
<td>20965</td>
<td>5076</td>
<td>3088</td>
<td>1736</td>
<td>4.13</td>
</tr>
</tbody>
</table>

8-connected grid

more beneficial for SR_GPU, since the propagation of pixel values is more effective for larger grids and, as a result, fewer raster scans are needed to achieve stability.

The performance analysis for FH_GPU under the scenario with input data size variation is also interesting. As larger input tiles are processed, maintaining the number of threads fixed, the algorithm increases its efficiency. The change from 4K×4K to 8K×8K tiles improved the throughput from about 1227 to 1533 millions of pixels visited and compared per second — using 8-connected grid and the GTX 580. The throughput increase is consequence of a better utilization of the GPU computing power, because of the smaller amortized synchronization costs at the end of each iteration. The speedup on top of the CPU version is also higher, since there is no throughput improvement for the CPU as the tile size increases.

For tiles larger than 8K×8K, no improvement in throughput is observed for FH_GPU either. Small increases in speedup values compared to the CPU version are observed for the 12K×12K and 16K×16K image tiles. These gains are consequence of the better performance achieved by the multiple iterations of the raster scan phase, which is the initial phase in FH_GPU before the queue-based execution phase. The raster scans have a better performance for larger image tiles because the overhead of launching the 4 GPU kernels used by the raster scan phase is better amortized since more pixels are modified per pass.

5 Conclusions

We have presented an implementation of a fast hybrid morphological reconstruction algorithm for GPUs. Unlike a previous implementation, the new GPU algorithm is based on an efficient sequential algorithm and employs a queue-based computation stage to speed up processing. Our experimental evaluation on two state-of-the-art GPUs and using high resolution microscopy images show that (1) multiple levels of parallelism can be leveraged to implement an efficient queue and (2) a multi-level queue implementation allows for better utilization of fast memory hierarchies in a GPU and reduces synchronization overheads.
Acknowledgments. This research was funded, in part, by grants from the National Institutes of Health through contract HHSN261200800001E by the National Cancer Institute; and contracts 5R01LM009239-04 and 1R01LM011119-01 from the National Library of Medicine, R24HL085343 from the National Heart Lung and Blood Institute, NIH NIBIB BISTI P20EB000591, RC4MD005964 from National Institutes of Health, and PHS Grant U1TR000454 from the Clinical and Translational Science Award Program, National Institutes of Health, National Center for Advancing Translational Sciences. This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH. We also want to thank Pavel Karas for releasing the SR_GPU implementation used in our comparative evaluation.

References

Optimization and Parallelization of the Matched Masked Bone Elimination Method for CTA

Henk Marquering, Jeroen Engelberts, Paul Groot, Charles Majoie, Ludo Beenen, Antoine van Kampen, and Silvia D. Olabarriaga

Dept. of Biomedical Engineering and Physics, Academic Medical Center, NL
Dept. of Radiology, Academic Medical Center, NL
SARA, NL
Dept. Clinical Epidemiology, Biostatistics and Bioinformatics, AMC, NL

Abstract. The fast radiologic evaluation of intracranial arteries in CT angiography data sets is hampered by the presence of the dense skull. An algorithm for the removal of the skull based on subtraction of additional CT images was therefore previously developed and deployed as a service in the Academic Medical Center in Amsterdam (matched masked bone elimination service). The method relies on a registration step that requires large computational time, which made this service unsuitable for acute care patient work up. We reduced the computation time by parallelizing the algorithm. The total time to generate an enhanced scan dropped from 20 min to below 10 min on 8 cores, enabling the usage of this service for the acute care. A proof-of-concept validation on 18 scans indicates that this speed-up has been achieved without quality loss, but further work is necessary to deploy the service in the hospital system.

Keywords: Parallelization, Acute care, Image analysis, 3D registration, neuroimaging, MPI, CTA

1 Introduction

Computed tomography angiography (CTA) is an established technique for the detection of vascular anomalies. It is less invasive than conventional angiography, more accessible than magnetic resonance imaging, and it can be performed within a short examination time. Therefore, CTA is currently the most frequently performed imaging procedure for diagnosis of vascular disease in acute care.

In maximum intensity projection (MIP) and volume rendering (VR) images of CTA scans, the arteries are often obscured by bone. The matched mask bone elimination (MMBE) technique has been introduced to remove bones from CTA source images [1] and refined in additional algorithm development [2, 3]. In Fig. 1 an example of the result of the MMBE algorithm is shown: on the left the intracranial arteries are obscured by the presence of the skull, which has a higher intensity in CTA images than the arteries. From this example it is clear that

* Corresponding author.
after applying the MMBE, the intracranial arteries can be easier assessed by a radiologist. MMBE is currently used in a number of clinical applications, for example detection of aneurysms [5] and venography [4].

MMBE uses an additional, non-contrast CT scan and creates a mask of the bone tissue by thresholding and dilation in order to remove the skull from 3D scans. The image analysis method runs on a dedicated server that is connected to the scanner console via the DICOM protocol. The current implementation is very time consuming, which prevents a broader use by the acute care. Whereas a typical run of the MMBE alone is currently performed in 8 to 20 minutes for a typical high-resolution CTA scan, in acute care the intracranial arteries need to be assessed within 10 minutes. This is the time window allowed by the protocol adopted in our hospital for critical treatment decisions in life-threatening situations. Moreover data transfers between the console and the MMBE server are also responsible for a delay in the overall time needed to prepare the enhanced scan for assessment by a radiologist. Therefore the MMBE method needs to run under 5 min to fulfill acute care requirements.

In this paper we describe the first steps taken to enable the porting of this service to acute care. The goal of this particular study was to achieve shorter execution times (less than 5 minutes) of the MMBE software using parallelization. We describe the design considerations (Sect. 2), the software optimization and parallelization strategies (Sect. 3), and a performance analysis and a proof-of-concept deployment (Sect. 4). We furthermore present preliminary conclusions and discuss the path ahead for clinical deployment (Sect. 5).

2 Background and Design Considerations

The MMBE method has a long history at the AMC. From its first publication in 2001 [1], the method has been refined with piecewise [2] and multi-scale [3] approaches. The method and its software implementation have been extensively validated for various clinical applications. Finally the software has been deployed as a service at a dedicated DICOM server connected to radiology CT scanners and workstations in 2009. Since then the MMBE service has been stable and used
routinely without modification around 3 times a week at the AMC for various clinical and research applications.

Figure 2 illustrates the main MMBE components: images are acquired by the CT scanner and transferred slice-by-slice to the vendor’s console. The complete scan is assembled and then transferred to the MMBE service using DICOM protocol. The image analysis is triggered automatically when the complete scans arrive at the MMBE service. Once the enhanced scan is ready, images are send back to the workstation that is nearest to the scanner that created the CTA images for viewing. The current server is exclusively used for healthcare and research applications at the AMC. Lab and IT technicians can access log files and status information using a simple web-interface.

Currently, however, the service is not useful for acute care because of large processing and image transfers time. A brief analysis of the algorithm indicated that speed-ups should be expected from a parallel implementation. Various alternatives were considered initially, for example usage of the clusters at the Life-Science Grid, local clusters, and multi-core approach. Usage of remote resources was discarded because of data privacy regulations that limit the processing to the trusted network of the AMC, in particular for acute cases when obtaining explicit consent is not possible. Moreover, the expectation was that one server with multiple cores would be enough to obtain the required speed-up, and that a shared memory parallelization approach could be adopted. Finally, we aimed at a solution where the number of parallel processes could be easily controlled such that the code could still run sequentially or on more modest hardware with few cores for applications in which there is no urgency. Given that the (clinical) validation of such a method and software can take significant amounts of effort, we also opted to change the algorithm as minimally as possible to minimize impact on the resulting enhanced scan.

3 Methods

The MMBE program runs roughly through the steps illustrated in Fig. 3: read data; pre-processing; slice-wise registration; masking off the bone tissue; and exporting results. MMBE starts with reading of the non-contrast CT and CTA
scans of the brain from the DICOM server. The slices are then prepared for registration by a sequence of filtering steps which mainly consist of sharpening the images. MMBE then registers the two scans by fitting the contrast to the non-contrast image using a slice wise rigid registration algorithm. Thresholding is performed on the blank scan to segment bone tissue. The resulting bone mask is applied to the contrast scan to remove bone pixels. Finally the enhanced scan is sent back to the CT workstation and it becomes available for viewing. Two separate applications handle communication with the DICOM server: DICOM C-STORE SCP (receive and store) and DICOM C-STORE SCU (transmit). More details about the method can be obtained in [1–3].

3.1 Initial conditions

MMBE is written in C++ and makes extensive use of the VTK library [10]. The Boost library [9] is used for directory handling routines, and the DCMTK library [11] for reading and writing DICOM files. The original implementation consisted of a VMWare virtual machine image that contained the necessary libraries and MMBE, both the source code and the binary, installed on Linux (Centos 5.5).

The total runtime for a sample scan inside the virtual machine varied between 15 and 20 minutes, depending on the hardware used. The main goal was to decrease the total runtime to less than 5 minutes to guarantee that the enhanced scan would be available for diagnosis within the target 10 min. Furthermore, the program had to run on a single multicore system.

3.2 Development and testing conditions

For developing and testing one of the nodes in the Dutch compute cluster (Lisa) hosted at SARA was used. The specifications for this system are: 2 Intel Xeon L5640 (2.26 Ghz) CPUs, 12 cores (2 x 6), 220 GB physical hard disk, 24 GB QPI 5.86 GT/s RAM, 12 MB Cache.

1 https://www.sara.nl/systems/lisa
The data used in this study was acquired with a 64-slice Siemens scanner. All slices consisted of 512x512 pixels with 16 bit grayscale, and the number of slices can vary for different patients. The average image was around 0.5 GB. Most of the tests and results described below are based on a high-resolution data set of 779 slices, which is referred to as the sample scan or image in the text below. Other two scans of low and medium resolution were also used in the performance assessment, with 178 and 333 slices.

3.3 Parallelization

The code was enhanced before the parallelization. The iteration order of a nested loop was modified to reduce cache misses, and the code was recompiled with another compiler. Originally MMBE and the VTK library were built with the GCC C++ compiler [8], but this was replaced by the Intel C++ compiler [12]. After these improvements the runtime dropped from 11 min 54 s to 5 min 38 s.

The analysis of the execution for the test dataset indicated that two thirds of the runtime is spent in the slice-wise registration. Only this step has been parallelized in this study, whereas the other steps have been kept intact to minimize the impact on the quality of results respectively to the original implementation.

During the registration a loop over all slices is performed, where each slice from the contrast scan is fitted onto the corresponding blank scan in an iterative manner. A good starting point leads to faster convergence per slice, therefore for each new slice, the transformation parameters of the previous one are used to initialize the registration. Due to the dependencies between the slices, this registration strategy prohibits the independent registration of separate slices in parallel. With this in mind, two types of parallelization were tested, a shared memory option (OpenMP®[6]) and a distributed memory option (OpenMPI [7]). In OpenMPI each process has its own data (variable) space, whereas in OpenMP all processes operate in the same data space in memory.

OpenMP. With OpenMP all processes use the same memory to access and store data, except when private memory space is explicitly defined for certain variables. Advantages of this approach are that the non-contrast CT, CTA, and resulting enhanced image, approximately 400 MB each for the sample data set, only need to be present once in memory. A second advantage is that no communication is needed between the separate processes. Unfortunately, the VTK library used in the MMBE implementation is not thread-safe because it uses static variables in some of its methods (functions). The modification of the VTK library would require effort outside of the scope of this project, so this approach was abandoned.

OpenMPI. Figure 4 illustrates the strategy adopted for performing the registration of slices in parallel. In the sequential version (left), the loop through the slices is split in two parts. Firstly, a loop from the middle of the dataset down is performed, from slice n/2 to 0. Secondly, a loop from the middle up (from slice
Fig. 4. Parallelized slice-by-slice registration algorithm for 1, 2 and 4 cores. Grey columns represent processes, and numbers indicate slice processing order. Arrows indicate transformation propagation to the next slice (straight arrows = broadcasting).

The transformation parameters from the nearest slice are propagated to the new set of slices (MPI broadcast). In the example, with 4 processes, 2 sets of slices are registered simultaneously: slices (n/2, n/2-1) and (n/2+1, n/2+2) are registered starting with the parameters from the preparation step. After all four processes are finished, the resulting transformation for slice n/2-1 is passed on to the upper two processes (n/2-2, n/2-3) and the transformation from slice n/2+2 is passed on to the lower slices (n/2+3, n/2+4). Note that we chose to keep the original approach of splitting the dataset in two chunks and starting from the center slice, instead of further splitting the dataset into more chunks when more processes are used. This is motivated by the knowledge that the center slice provides best initialization conditions for the registration process, and that changing it would require extensive additional validation.

In the adopted approach all processes read in the complete datasets and perform the same preparation for the slice-wise registration. Because they all run within the same node, the data needs to be cached only once. When all slices have been fitted, each process holds part of the results in its own (part of the) memory. These parts of the enhanced scan are collected on a single process, where serial execution of the subsequent steps is continued.
3.4 Proof-of-concept deployment

The parallel implementation has been deployed in a proof-of-concept setting at the AMC and applied to 18 datasets of consecutive patients with the following characteristics: pixel width and height of 150.00 mm; slice thickness of 1.3mm with increment of 0.5mm; 111 to 880 slices.

Two sets of tests were executed. The first were run on an Intel(R) Xeon(R) E5335 CPU (2.00GHz), 4 cores, 16 GB RAM, 4MB cache memory. This system is currently used for research data processing at the radiology department, therefore a realistic option for future deployment of the parallel MMBE service. In addition a faster computer was used for the same datasets: Intel(R) Core(TM) i7-3930K (3.20GHz) computer system, 2x6 cores with hyperthreading, 32 GB RAM, 12MB cache. We refer to these as the 4-core and the 6-core systems. Results were validated by visual inspection by an expert and compared with the enhanced scans calculated with the original MMBE service.

4 Results and Discussion

An initial performance assessment was performed for test images on the Lisa cluster. The code was instrumented to print wall clock time stamps after the completion of relevant stages.

Firstly, the overhead for loading the images in all processes was estimated. For the sample scan, reading of the data with the serial version of the software took 4.45 s, as compared to 5.73 s when the parallel version is run on 10 cores. Although the overhead of approximately 30% could be interpreted as rather large, an increase of less than 1.5 s was considered negligible compared with the total execution time. We observed however a drawback of duplicating the complete images on all processes, because much more memory is used than when the data is read on a single process and then distributed in parts to others via communication. In practice memory capacity turned out to limit the number of parallel processes. For example a physical memory of 24 GB supports a maximum of 10 parallel processes for the sample dataset and 12 processes for smaller ones.

Secondly, we measured the processing time of the MMBE method excluding data transport times: the total time to generate the enhanced image and to perform the registration step on the three test datasets. Table 1 presents the obtained results. As expected the longest times have been measured for the high resolution scan, from 335.9 s (5.6 min) on a single core to 144.2 (2.4 min) on 10 cores. Lower resolution scans are processes in less than 2 min. These results are well below the timings measured at the initial conditions (15-20min).

Speedup factors have been calculated for the registration step only, respectively to the single core performance – see Tab. 1 and Fig. 5. Speedup factors range from 1.7 up to 10 using 2 to 12 cores, which represents 80% of the ideal speedup. This trend is observed throughout the high, medium and low resolution test datasets. Note however that more gain is obtained when an even number of processes is used. When an odd number of processes is used, the amount
Table 1. Performance results for 3 test images: number of cores, execution time for complete MMBE and registration (seconds), percentage of registration/total time, and speed-up factor (gain) for registration respective to single core execution.

<table>
<thead>
<tr>
<th>n</th>
<th>high (779 slices)</th>
<th>medium (333 slices)</th>
<th>low (178 slices)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>reg</td>
<td>% gain</td>
</tr>
<tr>
<td>1</td>
<td>335.9</td>
<td>227.2</td>
<td>67.6</td>
</tr>
<tr>
<td>2</td>
<td>243.3</td>
<td>134</td>
<td>55.1</td>
</tr>
<tr>
<td>3</td>
<td>243.5</td>
<td>131.2</td>
<td>53.9</td>
</tr>
<tr>
<td>4</td>
<td>173.2</td>
<td>64.6</td>
<td>37.3</td>
</tr>
<tr>
<td>5</td>
<td>160.4</td>
<td>50.5</td>
<td>31.5</td>
</tr>
<tr>
<td>6</td>
<td>160</td>
<td>46.5</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>159.7</td>
<td>44.1</td>
<td>27.6</td>
</tr>
<tr>
<td>8</td>
<td>144.9</td>
<td>37</td>
<td>25.5</td>
</tr>
<tr>
<td>9</td>
<td>144.2</td>
<td>37.5</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>146.8</td>
<td>29.6</td>
<td>20.2</td>
</tr>
<tr>
<td>11</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>12</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

of work is not equally split between them, so the "slower" process limits the performance (e.g. running on 5 processes is similar to running on 4 processes performance-wise).

Finally, Tab. 2 presents the total time to generate the enhanced scan using the 4- and 6-core systems for 1 to 4 cores. With both systems the average execution time is below the target of 5 min. The first qualitative assessment of results indicate that the quality is similar to the original method. For one image the program presented an error on the 4-core system, which is currently being investigated. We also executed the method on 5 to 12 cores on the 6-core system (see Fig. 6), however the average execution time actually increases on more than 6 cores, indicating that hyperthreading does not work well for this application.

5 Conclusions

We presented the rationale used to reduce the execution time of the MMBE method to enable its usage in an acute clinical setting. We chose for a simple approach, keeping the original method as intact as possible. This prevented us from taking advantage of the large amount of literature on the topic of parallel registration of medical images, for example [13, 14].

The presented results show that the goal of running it under 5 minutes has been achieved. The parallelization proved to be helpful, but also the code optimizations and usage of better compiler helped to reduce the execution time to the desired target. In the new 6-core system, the target can actually be reached with a single core.

Results are promising for usage in acute care, however additional steps still need to be taken before deployment in clinical practice. Firstly, the quality of
Fig. 5. Speed-up obtained with the parallel MMBE on 1-12 cores for three test scans of high, medium and low resolution, respectively with 779, 333 and 178 slices.

Fig. 6. Average execution time on the 18 datasets using the 6-core system.

results for clinical usage need to be confirmed in a more rigorous study; however, because the modifications to the MMBE software were kept to a minimum, we expect this validation to be straightforward. Secondly, the data transfer between the scanner and the MMBE service needs to be optimized to avoid unnecessary waiting for DICOM images to be assembled. And finally, the software needs to be certified before it can be deployed in the radiology IT systems.

Acknowledgments

This work was supported by the Netherlands Organisation for Scientific Research (NWO; Programma parallelisatie, NRG-2010.05). Moreover we thank Y. Mohammed for his early contribution to this project; W. Vermin and W. de Jong from SARA for their assistance; H. Gratama van Andel for details about the MMBE software, and M. Brandt for setting up the 6-core system.

References

1. H.W. Venema, Hubmans FJ, den Heeten GJ, CT angiography of the circle of Willis and intracranial internal carotid arteries: maximum intensity projection with
Table 2. Time (seconds) to run MMBE in various configurations for 18 datasets.

<table>
<thead>
<tr>
<th></th>
<th>4-core system</th>
<th></th>
<th></th>
<th>6-core system</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 core</td>
<td>2 cores</td>
<td>4 cores</td>
<td>1 core</td>
<td>2 cores</td>
<td>4 cores</td>
</tr>
<tr>
<td>1</td>
<td>2.92</td>
<td>2.49</td>
<td>1.90</td>
<td>1.10</td>
<td>0.96</td>
<td>0.75</td>
</tr>
<tr>
<td>2</td>
<td>4.33</td>
<td>3.57</td>
<td>3.11</td>
<td>1.70</td>
<td>1.39</td>
<td>1.16</td>
</tr>
<tr>
<td>3</td>
<td>4.10</td>
<td>2.79</td>
<td>2.78</td>
<td>1.58</td>
<td>1.12</td>
<td>1.05</td>
</tr>
<tr>
<td>4</td>
<td>4.87</td>
<td>3.29</td>
<td>2.64</td>
<td>1.80</td>
<td>1.29</td>
<td>0.99</td>
</tr>
<tr>
<td>5</td>
<td>5.01</td>
<td>3.75</td>
<td>3.17</td>
<td>1.95</td>
<td>1.45</td>
<td>1.13</td>
</tr>
<tr>
<td>6</td>
<td>6.17</td>
<td>4.89</td>
<td>-</td>
<td>2.44</td>
<td>1.82</td>
<td>1.52</td>
</tr>
<tr>
<td>7</td>
<td>2.22</td>
<td>1.75</td>
<td>1.46</td>
<td>0.85</td>
<td>0.69</td>
<td>0.56</td>
</tr>
<tr>
<td>8</td>
<td>5.17</td>
<td>3.76</td>
<td>3.06</td>
<td>2.02</td>
<td>1.46</td>
<td>1.12</td>
</tr>
<tr>
<td>9</td>
<td>2.56</td>
<td>1.80</td>
<td>1.53</td>
<td>0.99</td>
<td>0.73</td>
<td>0.62</td>
</tr>
<tr>
<td>10</td>
<td>4.54</td>
<td>3.25</td>
<td>2.65</td>
<td>1.76</td>
<td>1.22</td>
<td>0.96</td>
</tr>
<tr>
<td>11</td>
<td>4.98</td>
<td>3.62</td>
<td>3.00</td>
<td>1.94</td>
<td>1.39</td>
<td>1.09</td>
</tr>
<tr>
<td>12</td>
<td>5.24</td>
<td>3.86</td>
<td>3.00</td>
<td>2.03</td>
<td>1.50</td>
<td>1.12</td>
</tr>
<tr>
<td>13</td>
<td>5.71</td>
<td>4.16</td>
<td>3.59</td>
<td>2.22</td>
<td>1.58</td>
<td>1.26</td>
</tr>
<tr>
<td>14</td>
<td>5.15</td>
<td>3.72</td>
<td>3.17</td>
<td>1.99</td>
<td>1.42</td>
<td>1.14</td>
</tr>
<tr>
<td>15</td>
<td>1.97</td>
<td>1.56</td>
<td>1.33</td>
<td>0.77</td>
<td>0.61</td>
<td>0.51</td>
</tr>
<tr>
<td>16</td>
<td>2.44</td>
<td>1.91</td>
<td>1.65</td>
<td>0.93</td>
<td>0.75</td>
<td>0.62</td>
</tr>
<tr>
<td>17</td>
<td>2.40</td>
<td>1.82</td>
<td>1.62</td>
<td>0.91</td>
<td>0.72</td>
<td>0.63</td>
</tr>
<tr>
<td>18</td>
<td>2.62</td>
<td>1.98</td>
<td>1.70</td>
<td>1.01</td>
<td>0.78</td>
<td>0.65</td>
</tr>
</tbody>
</table>

6. The OpenMP® API specification for parallel programming http://openmp.org/
11. DICOM ToolKit (DCMTK) package, http://support.dcmtk.org/docs/
Semantic Federation of Distributed Neurodata

Alban Gaignard¹, Johan Montagnat¹, Catherine Faron Zucker¹, and Olivier Corby²

¹ I3S laboratory, CNRS & University of Nice Sophia Antipolis, France
http://www.i3s.unice.fr
² INRIA Sophia Antipolis, France
http://www-sop.inria.fr

Abstract. Neurodata repositories federation is increasingly needed to
implement multi-centric studies. Data federation is difficult due to the
heterogeneous nature of distributed data repositories and the technical
difficulties faced in addressing simultaneously multiple data sources. This
paper describes a data federation system based on a semantic mediation
layer and an advanced distributed query engine able to interface to mul-
tiple heterogeneous neurodata sources. Both performance and usability
indicators are shown, demonstrating the soundness of the approach and
its practical feasibility.

1 Introduction

Computational neurosciences are highly dependent on the availability of large
neurological datasets. An increasing effort has been invested in distributed comput-
ing infrastructures to face the challenges raised by modern multi-centric neu-
roscience studies [1,2,3]. In particular, distributed neurodata federation is con-
sidered to harness the large neurodata resources available [4,5,6]. Federation of
heterogeneous data sources is a complex problem due to the need to align the
different data models used, and the lack of tools to efficiently query and process
distributed databases. The Semantic Web community has developed methodolo-
gies and tools to clearly define the semantics of data using models and computer-
interpretable metadata. This paper describes how these approaches can be ap-
pplied to neurodata alignment, and introduce new methods to integrate legacy
neurodata sources in a federated platform.

An approach often considered to deliver federated data repositories is data
warehousing which consists in statically importing all data sources in a cen-
tralized repository after semantic alignment. Although simple, this solution has
many drawbacks such as the need to transform legacy data entities to different
representations, the scalability of the centralized repository, the central point of
failure thus created, etc. When considering medical data, it is often even not
possible to create a central data repository for legal and ethical reasons. The
alternative solution studied in this paper is a dynamic distributed data federa-
tion mechanism which aligns the data models and the query capability of the
heterogeneous repositories to deliver a unified view over the complete data set.
This approach pushes queries to the distributed data sources and gathers results. From the client point of view, distributed data sources are thus virtually integrated as if a single data source was queried. It raises new issues such as the need for expressing queries that can be mapped to multiple data sources, network communication overhead, coherency of distributed queries, and transforming data entities on-the-fly to different representations while preserving their semantics.

We developed the KGRAM distributed query system to align multiple neurodata sources semantically and map queries to the heterogeneous federation. System utilisability is considered by taking into account both the query language expressiveness and performance issues. KGRAM is generic in the sense that it does not require any prior knowledge on the data sources content, and robust in a distributed environment as it transparently adapts to topology changes caused by the addition or the removal of data sources.

2 Distributed data query engine

Different neurodata repositories generally use different data models and store heterogeneous data. The dynamic data federation approach addresses simultaneously the problems of data source heterogeneity and data distribution. It relies on a central federator, and a set of federated data providers. From a unique query, the federator is responsible for the coherent sub-querying of the federated data providers and for unifying all results found into a global result set. It avoids consistency and synchronization issues generally observed in data warehouses where data transformation is performed periodically.

A single main query language is needed to express federation-wide queries. The SPARQL\(^3\) language v1.1 is considered in this work as a highly expressive, versatile, and de facto standard semantic data query language. Semantic data is viewed as a collection of triple patterns formalized as \((s,p,o)\), where a subject \(s\) is linked to an object \(o\) through a predicate \(p\), and usually represented as RDF triple\(^4\). A collection of triples constitutes a knowledge graph. Semantic querying through SPARQL is equivalent to the matching of a graph-based query pattern into the complete knowledge graph.

More precisely, the SPARQL-based query engine KGRAM described below (i) transforms a graph-based semantic query into the target data sources query languages and (ii) combines the triple results to assemble the reply at runtime. To efficiently perform distributed queries, the engine implements several optimizations including query rewriting and intermediate results exploitation.

2.1 The KGRAM query engine

KGRAM stands for Knowledge Graph Abstract Machine \(^7\). It is a versatile system aiming at representing, querying and reasoning over semantic data represented as Knowledge Graphs \(^8\). It enables querying different data source

\(^3\) SPARQL: [http://www.w3.org/TR/rdf-sparql-query/](http://www.w3.org/TR/rdf-sparql-query/)

\(^4\) RDF: [http://www.w3.org/TR/rdf-primer/](http://www.w3.org/TR/rdf-primer/)
models, provided that they are able to produce a graph view of their data, e.g. RDF but also XML or relational databases. For the sake of generality, KGRAM introduces a set of abstract operators manipulating abstract graph data structures (abstract Nodes and Edges forming abstract Graphs). Graphs are navigated through so-called Producers responsible for the iteration over their Nodes and Edges. For each data source kind, a specific Producer is needed that abstracts its representation to answer SPARQL queries by returning data represented as graph elements. Knowledge graphs are matched with queries through Matchers. Depending on their implementations, the comparison may consist in simple labels equality or it may take into account domain knowledge through class and property subsumption (RDS entailment) or eventually approximate matching based on semantic similarities. Finally, Filter operators are responsible for evaluating value constraints over knowledge graphs.

2.2 Distributed query processing

In KGRAM, the evaluation of SPARQL queries basically consists in searching for matching edges in a knowledge graph delivered by a Producer. To mashup linked data over distributed data sources, KGRAM introduces the notion of Metaproducer responsible for the enumeration of Nodes and Edges coming from several Producers interrelated to several data sources. All producers implement the same query interface, receiving the same messages from the Metaproducer and rewriting the input queries into native data source ones, to shield KGRAM core from data source heterogeneity. To handle distributed data repositories, we extended KGRAM implementation by providing a web service implementation of its Producer interface. Figure 1 illustrates the main elements involved in the semantic distributed query processing.

Web service endpoints are queried over the standard SOAP protocol. Each remote Producer interfaces with its database. It is responsible for enumerating on-the-fly triples matching the SPARQL queries received. Various Producer implementations enable querying various data models. The ParallelMetaProducer
is responsible for the exploitation of service parallelism and for merging triples coming from the distributed data sources into the resulting knowledge graph. Algorithm 1 illustrates how the ParallelMetaProducer distributes a SPARQL query over a set of Producers.

**Algorithm 1:** Naive parallel distributed query processing, with an explicit wait condition.

**Data:** Producers the set of SPARQL endpoints, 
TriplePatterns the set of triple patterns forming a basic graph pattern in the initial SPARQL query, 
scheduler a thread pool allowing for parallel execution.

**Result:** Results the set of SPARQL results.

1. foreach \( (tp \in \text{TriplePatterns}) \) do
2.  foreach \( (p \in \text{Producers}) \) do in parallel
3.  scheduler.submit(\(p.\text{matchingTriples}(tp)\)) ;
4.  while (not scheduler.isFinished()) do // synchronization barrier
5.  foreach (task \( \in \text{scheduler.getFinished()} \)) do
6.  Results ← scheduler.getTask().getResults() ;

The principle of the algorithm consists in iterating over each triple pattern forming the initial SPARQL query (line 1). For each triple pattern, all remote Producers are queried concurrently (line 3). The federator then waits for all federated endpoints to finish through a synchronization barrier (line 4). Results are finally accumulated for the current triple pattern (lines 5 and 6) and the next triple pattern iteration can be processed (line 1). To soften the synchronization barrier a pipelining strategy was implemented, in which a synchronized blocking queue allows the federator to post-process results as soon as they become available. Due to space constraints, the pipelining algorithms are not detailed.

### 2.3 Distributed query optimization strategies

In KGRAM, the query distribution principle consists in iterating over each triple pattern request occurring in the SPARQL query. Each triple pattern request is dynamically wrapped into a unitary SPARQL query and pushed to remote data sources. Resulting triples are returned back and accumulated into the KGRAM result graph. To enhance the efficiency of distributed query processing, a set of static and dynamic optimizations were implemented.

The filter pushing strategy consists in analyzing the initial SPARQL query to extract value constraints expressed in FILTER clauses. When iterating over each triple pattern request, each applicable value constraint is extracted from the initial FILTER and propagated as a new FILTER clause added to the triple
pattern query. This reduces drastically the communication cost since it prevents
the federator from transferring irrelevant results that would be finally filtered.

The same idea is exploited at query runtime through bind joins. The KGRAM
query processor manages an index of already known mappings between variables
and values, thus forming intermediate results. This cache is exploited at query
runtime in order to dynamically replace triple pattern variables by their associ-
ated values in the queries pushed to federated endpoints. Similarly to the filter
pushing strategy, bind joins reduce drastically the size of transferred results.

2.4 Relational data sources

In neurosciences, it is common that the underlying legacy environments rely on
traditional relational databases. KGRAM comes with a default implementation
of its Producer interface for RDF sources. To cope with relational sources, a
specific Producer that rewrites triple patterns forming the initial SPARQL query
into SQL queries is needed. Query results can then be mapped to the variables of
the original SPARQL query to build result graph triples. A generic SQL Producer
is out of scope of this paper. An ad-hoc Producer implementation was considered
in the case of the NeuroLOG platform described below.

3 Experimentation in the NeuroLOG platform

The NeuroLOG platform federates data and computational resources from
collaborating neuroscience centers. The prime objective of NeuroLOG is to adapt
non-invasively to the legacy environments deployed in each participating center,
so that each site remains autonomous in the management of its internal resources
and tools, while benefiting from the multi-centric medical studies support from
the middleware. The platform is federating 5 neuroscience centers spread over
France. On each site, the data source is composed of raw neuroradiology files, and
description metadata linked to these files (information on image data acquisition,
data content, neuropsychological tests associated to images, etc) [6]. The source
metadata is often represented and managed in relational databases for historical
reasons.

Blue components in Figure 2 sketches the architecture of the NeuroLOG mid-
dleware. On each neuroscience site an independently managed legacy relational
database is deployed. It is completed by a NeuroLOG middleware database. The
multi-centric studies conducted by neuroscientists may be perceived under sev-
eral facets, involving both the native relational data representation and a seman-
tic data representation enabling richer queries. The DataFederator commercial
tool is used to dynamically federate relational data sources into a unified view.
It can perform SQL queries that are distributed over all platform data sources.
It includes both a mediation layer that aligns heterogeneous relational databases

schemas, and rewrite SQL queries applying to the federated view to match the various source schemas. The data mediation semantic alignment is based on a domain ontology, called OntoNeuroLOG that was developed in the context of this project. A federated relational schema is derived from OntoNeuroLOG, serving as the federated view schema. In addition to this relational representation, a semantic representation of the same data sources was created to enable richer querying features delivered by Semantic Web query engines. A centralized approach was adopted, where all relational data sources are mapped to RDF triples (using the MetaMORPHOSES tool\(^6\)) and aggregated in a unique semantic repository. The NeuroLOG platform thus exposes a dual view of the federation metadata, enabling both dynamic SQL querying and static SPARQL querying.

![Fig. 2. Data management layer of the NeuroLOG platform](image)

Although very flexible, this system is also confusing for end users due to the dual view of all data entities. The semantic repository is also subject to limitations of a static, centralized warehouse. To overcome these issues, the NeuroLOG platform was extended with the KGRAM query engine introduced in section 2.1 (see green components in Figure 2). A KGRAM remote producer was deployed on top of each site legacy database. This endpoint exposes the site data content in RDF through its Producer. Depending on the site deployment option, it either interfaces dynamically to the site native relational database (option 1 in Figure 2), or accesses an RDF repository representation of this legacy database (option 2) which is periodically regenerated using MetaMORPHOSES. Consequently, NeuroLOG’s centralized RDF repository (crossed over in red in Figure 2) is not needed anymore. It is replaced by a completely dynamic semantic federation (all sites using 1), a distributed version periodically updated (all sites using option 2), or a mixture of both. This setting enables the unified querying

\(^6\) MetaMORPHOSES: [http://metamorphoses.sourceforge.net](http://metamorphoses.sourceforge.net)
of the RDF repositories and the platform legacy relational databases through
the SPARQL language. It solves the central repository limitation, distributing
the query load over the federation data servers. It proposes a single view over all
data. The experiments reported below demonstrate the query expressiveness and
the performance of the KGRAM engine. The NeuroLOG platform both provides
a real use case for these experiments and serves as a performance measurement
reference.

3.1 Experimental setup

Three query environments are compared. The relational federation only exposes
heterogeneous relational databases, virtually integrated through the DataFeder-
ator commercial middleware. It corresponds to the seminal NeuroLOG platform
deployment. Two other environments (semantic federations) expose heteroge-
neous data sources virtually integrated through the KGRAM framework. In the
RDF semantic federation environment, all sites are configured with option 2.
It corresponds to a modification of the NeuroLOG platform where the central
semantic repository is spread over all participating sites but native repositories
are transformed to RDF before being accessed. In the RDF+SQL semantic fed-
eration environment, a mixture of RDF sources and SQL sources (dynamically
accessed through 1) is used to evaluate a completely heterogeneous set up.

Figure 3 (top) illustrates a real clinical use case involving querying neurodata
distributed over several centers. It aims at searching for datasets (acquired with
Gadolinium contrast agent) associated to patients (join performed at line 4) in
the context of multi-centric studies addressing the Multiple Sclerosis pathology.
During the evaluation of this query, all types of medical images will be searched
in order to match a “GADO” tag (indicating use of the Gadolinium contrast
agent). But the evaluation of this query can be considerably enhanced by ex-
ploring clinical knowledge represented in an RDFS vocabulary. Indeed Gadolin-
ium is only used in the context of the MR modality. By exploiting this domain
knowledge with KGRAM, the query time can be significantly reduced since all
non-MR datasets are excluded. This results from the triple pattern (dataset rdf:type
dataset:MRDataset) added to the query which leads, under RDFS entailment, to less
intermediate results to be transferred and thus an overall faster distributed query
evaluation.

The bottom query from Figure 3 achieves the same clinical objective but
makes use of SPARQL property path expressions. This language feature aims
at representing paths between two resources by only specifying, in the form
of patterns, the sequence of mandatory, optional, reverse, or multiple repeti-
tion of properties linking the resources together. It brings a high expressivity to
SPARQL queries and it is particularly adapted in the context of graph-based
querying. This kind of query cannot be easily expressed in SQL. It is thus diffi-
cult to implement it with traditional relational databases. As a full SPARQL 1.1
interpreter, compliant with property path expressions, KGRAM allows for per-
forming this kind of graph-based information retrieval against SQL data sources,
which would not have been possible wit traditional SQL query engines.
3.2 Qualitative evaluation

KGRAM implements the RDFS entailment regime (subsumption between classes or properties), and other inferences based on algebraic properties like the transitivity or symmetry of properties and inverse properties. It thus provides a richer query interface to legacy databases participating into the federation. In addition, in the context of collaborative platforms exposing legacy relational databases, we argue that the design of queries is more intuitive through knowledge-based languages such as SPARQL than through traditional relational languages such as SQL. Indeed, the navigation through links between entities is explicit in SPARQL while it is implicit in SQL and generally requires for intermediate joins. The following SQL query corresponds to the SPARQL query of Figure 3:

```sql
SELECT Subject.subject_id, Subject.subject_common_identifier, Dataset.name
FROM Study, Subject, Dataset, Rel_Subject_Study
WHERE Rel_Subject_Study.Subject_subject_id = Subject.subject_id AND
Rel_Subject_Study.Study_study_id = Study.study_id AND
Dataset.Subject_subject_id = Subject.subject_id AND
Subject.subject_common_identifier LIKE '%MS%' AND Dataset.name LIKE '%GADO%'
```

Whereas joins are naturally expressed in the SPARQL query (line 4 of Figure 3), it is not the case in SQL since a join table may be needed (Rel_Subject_Study table, line 3) and must be explicit (line 4, 5 and 6). This definitely complicates the query design, generally considered as complex, error-prone, and time consuming.

3.3 Quantitative evaluation

The distributed query processing times of the relational federation (using DataFed- erator) and both semantic federations (RDF and RDF+SQL) are compared using KGRAM through two queries. Query Q1 corresponds to Figure 3 and query...
Q2 searches for datasets acquired through the T2-weighted MRI modality. Q2 leads to only 5 results and is thus, a very selective query. To be robust against variability observed in real distributed systems, results are averaged over three query runs. The average query execution time \( \pm \) one standard deviation is displayed in the following table showing that for Q1, leading to 336 remote invocations, the query times are lower with the optimized SQL federation engine DataFederator than with the semantic federation, but it remains in the same order of magnitude. For very selective queries such as Q2, we observe comparable query times for all environments:

<table>
<thead>
<tr>
<th></th>
<th>Semantic federation: RDF or RDF+SQL</th>
<th>Relational federation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 (s)</td>
<td>6.13 ± 0.05</td>
<td>11.76 ± 0.05</td>
</tr>
<tr>
<td>Q2 (s)</td>
<td>0.60 ± 0.03</td>
<td>1.53 ± 0.14</td>
</tr>
</tbody>
</table>

4 Concluding remarks

Transparent data federation approaches generally address performance issues. For instance, DARQ [10], SPLENDID [11] or FedX [12] propose a set of static and dynamic optimizations. KGRAM implements similar strategies to exploit value constraints at query rewriting time and to enhance the querying at runtime through bind joins. Moreover, in line with FedX, KGRAM also exploits the parallelism of both distributed data sources and modern multi-core CPUs with a multithreaded implementation. There remain other optimization opportunities for KGRAM such as data sources selection and subsequent query planning techniques, but to the best of our knowledge, none of the state-of-the-art transparent federating approaches address both the issues of efficient and heterogeneous distributed querying. The strength of the KGRAM model is to provide an extensible framework allowing to efficiently and dynamically access to multiple query sources internally using different data representations. In the future, the query rewriting capability of KGRAM is also thought to be a mean to implement fine-grained access control to data sources.

KGRAM eases the federation of distributed and heterogeneous neurodata repositories. Concrete examples in the context of the NeuroLOG platform, mixing both semantic and relational repositories, demonstrate the feasibility and the efficiency of this approach. Beyond performance, the query environment is enriched by the high expressivity of semantic query language. End-users gain from using knowledge with a well-defined semantics. Moreover, through an ontology and its associated entailments, the querying process can exploit domain knowledge to implement smart queries. The current implementation of KGRAM enables read-only access to data sources. Modifying heterogeneous distributed data sources is a challenging problem that we intend to tackle in the future.

Acknowledgments

This work was partly funded by the French National Agency for Research under grants ANR-06-TLOG-024 (NeuroLOG project).
References


Content-based Image Retrieval on Imaged Peripheral Blood Smear Specimens using High Performance Computation

Xin Qi¹,², Fuyong Xing³, Meghana Ghadge³, Ivan Rodero¹, Moustafa Abdelbaky³, Manish Parashar³, Evita Sadimin¹,², David J. Foran¹,²,⁴, Lin Yang⁵

¹ Dept. of Pathology and Laboratory Medicine, UMDNJ-Robert Wood Johnson Medical School
² Center for Biomedical Imaging & Informatics, The Cancer Institute of New Jersey
³ NSF Cloud and Autonomic Computing Center & Rutgers Discovery Informatics Institute, Dept. of Electrical and Computer Engineering, Rutgers University
⁴ Dept. of Radiology, UMDNJ-Robert Wood Johnson Medical School.
⁵ Division of Biomedical Informatics, Dept. of Biostatistics, Dept. of Computer Science, University of Kentucky

Abstract: Research in content-based image retrieval (CBIR) has emerged as an important focus of investigation multiple image-related disciplines. In this paper, we demonstrated our CBIR system on digitized peripheral blood smears using three different image sets. The system features white blood cell identification, and three-stage searching using a hierarchical annular histogram (HAH) approach. The results in this paper show that using the most conservative searching strategy, all the query images (up to 925 images) under 10x, 20x, and 40x magnification objectives have been correctly retrieved from the databases and a wide range of different white blood cell types including basophil, eosinophil, lymphocyte, macrophage, monocyte, and neutrophil can be correctly detected and retrieved as well. We utilized the CometCloud autonomic Cloud engine to run the CBIR algorithms in parallel over a database of images using Rutgers University’s resources in combination with federated High Performance Computing (HPC) cyber-infrastructure, Grids and Clouds. The results showed that the CBIR strategy that we have developed can be successfully executed online (in minutes) as compared to weeks which would be required using standard computing solutions.

Keywords: Content-based image retrieval, blood smear, high performance computation

1. Introduction:

Content-based image retrieval (CBIR) has been one of the most active research areas in a wide spectrum of image-related fields over the last few decades[1]. The application areas include web searching, networking for image storage and transmission, natural image processing and biomedical research. In the medical field, hundreds and thousands of digital images are routinely generated every month in ever-increasing quantities for diagnosis and therapy. Besides the steady growing rate of image production, clinical decision support techniques such as case-based reasoning[2] and/or evidence-based medicine[3] are creating a compelling need for reliably retrieving images leading to support for a variety of different diagnostic decisions.

Many of CBIR systems existed since the 1980s including some well-known examples of using simple color and texture characteristics to describe image contents[4-7] and higher level diagnostic related information for image queries[8, 9]. Typically CBIR contains four phases including content localization, feature extraction, performance evaluation and practical usability. The characteristics features from image may include intensity and shape[10], color[11-13], texture[14, 15] or any combination[12, 16] of these. The retrieval results are usually rank-ordered by some criteria, such as appearance similarity, diagnostic relevance et al. One of the obstacles to use CBIR in medicine is the lack of effective representation of medical content by low-level mathematical features[17].

There are many application fields in diagnostic medicine and three related domains would benefit tremendously from the use of content-based access methods including teaching, investigative research and therapy planning. CBIR has been implemented or proposed in various research areas such as radiology, pathology and dermatology and cytology.

In pathology, hematology already contains a large number of tools to automatically count blood cells. To classify abnormal white blood cells and compare diagnosis between a new case and cases with similar abnormalities is an interesting application. In this paper, we mainly focused on CBIR on digitized peripheral blood smear specimens using low-level morphological features.

In this paper, we present the use of CometCloud to execute CBIR in parallel on multiple HPC and Cloud resources with the goal of reducing the completion time. CometCloud is an autonomic Cloud engine built on top of a robust and scalable overlay, which provides programming platforms (MapReduce, Work-
flow, Master-Worker/BOT) that run on dynamically federated HPC, Grid and Cloud infrastructure. In previous work, the integration of public/private clouds and autonomic cloud bursts[18], i.e., dynamic scale-out to clouds to address dynamic workloads, spikes in demands, and other extreme requirements was explored[19-21]. However, due to the computational power required to run CBIR online, in order to provide answers within minutes rather than weeks, innovative solutions involving massively parallel systems (e.g., HPC resources or GPUs) must be explored.

Clouds are rapidly joining high performance computing systems, clusters and Grids as viable platforms for scientific exploration and discovery[22]. Hence, we exploit the potential massive parallelism of the problem (i.e., more than 46,000 image processing tasks) by combining the resources at Rutgers with distributed HPC cyber-infrastructure and Clouds using CometCloud. With such an approach, HPC resources can be elastically complemented with Clouds, for example during queuing periods.

2. Material and Method

This work is an extension of the Pathminer[23] project. The goal of this work is to find specific blood cell specimens in the query sample to support the diagnosis. To simulate the process of pathologist searching for various types of white cells, three sets of blood smear image data sets taken under three magnifications (10x, 20x and 40x objectives) respectively were investigated. Lower magnification (10x and 20x) give a gross overview of the whole image to quickly identify the region of interests and high magnification (40x) provides sufficient resolution for different types of blood cells. There were 925 blood smear images (1000x1000 pixels) taken under 10x magnification objectives; 96 images taken under 20x magnification objectives and 25 images taken under 40x magnification objective. There is no any correlation among those images under three different magnifications. Within the paper, we are mainly focused on four phases: white blood cells identification, three-stage searching by hierarchical annular histogram (HAH), performance evaluation and practical usability (which is achieved by running CBIR on CometCloud).

2.1 Phase 1: White cell identification

For the blood smear study, our focus is on retrieving various classes of white blood cells, which have blue nuclei stain in contrast with red blood cells for which this feature is absent. Therefore, using color-decomposition and morphology processing, centers of all types of white cells were identified as regions of interest. These retrieved regions of interests are served as the center of the searching regions for the following steps in the feature extraction phase.

2.2 Phase 2: Three-stage searching by HAH

**HAH feature extraction:** To quantify an effective representation of medical content by low-level mathematical features, we had proposed a novel feature measurement called hierarchical annular histogram (HAH). HAH is rotation invariant and can capture the spatial configuration of the landmarks within the image patch. Figure 1 illustrates the calculation of the proposed hierarchical annular histogram (HAH). Besides the central rectangle, within the rest of each rectangle-ring, intensity histogram RGB channels were calculated and concatenated together as the feature vector. Because HAH takes into consideration of the spatial configuration of the features, it can differentiate images with similar total intensity distribution, but different spatial intensity configurations. Figure 2 shows an example of two image patches with similar whole patch traditional color histograms but different HAH histograms.

**Three-stage searching:** The searching process of our CBIR consists of three stages: rough searching using hierarchical approach with HAH, refined searching by computing color histogram from 8-equally-divided segments of each rectangular annular bin from the results of rough searching, and mean-shift clustering for final results.

Stage 1 is rough searching. The algorithm begins with calculating the histograms of the most central bins for candidate image patches and comparing them with that of the query patch. Based on similarity of Euclidean distance calculation to query image, from the ranked similarity, 50% of candidates having less similarity were discarded. Sequentially, it only calculates the HAH from the next adjacent outside rings of the left 50% of candidates from the previous step, and then 50% of the left candidates ordered by ranked similarity to query patch were discarded. This step continues until it reaches the most outside ring. The scheme of hierarchical searching can greatly reduce the computational time, because each iteration decreas-
es the searching candidates significantly. After a few times iterations, only a few candidates with higher ranked similarity to the query image were preserved.

Stage 2 is refined searching. Within the remaining candidate images, each rectangular annular bin for both query patches and the candidate patches were equally divided into eight segments, and histogram was calculated from each segment within the central and ring regions. The final candidates are chosen based on the Euclidean distance of the concatenated color histograms. This stage is designed to delete the less similar patches from the first stage to improve the final retrieval results. This stage is not time consuming due to the limited number of candidates left from the first stage search.

Stage 3 is the final mean-shift clustering [24]. Because of the high correlation among those adjacent patches, mean-shift clustering was applied on the top 10% ranked patches to provide the most similar one to query patch. The bandwidth of mean-shift algorithm is selected as

\[
\text{Bandwidth} = \frac{\sqrt{\text{width of Query Patch}^2 + \text{height of Query Patch}^2}}{2}
\]

![Hierarchical Annular Histogram (HAH)](image)

**Fig. 1.** An illustration of HAH calculation. The left bottom plot is the histogram of the central ring; the right bottom plot is the histogram of the fourth ring counting from center.

### 2.3 Phase 3: Performance Evaluation

Patch retrieval is executed on RGB blood smear images with multiple searching parameters. The retrieval performances were evaluated based on whether the query images were correctly ranked as the No. 1 searching image patch within the database. Different types of white blood cells (at 20x) were evaluated and parameters are listed as follows:

**M:** magnification of different imaging objectives. M is 10x, 20x and 40x magnification in this study.

**P:** The percentage of the overlapping regions during searching P changed from 90% to 80%, 70%, 60% and 50%.

**R:** number of outside rings besides one central rectangular region. R changed from 9 to 7, 5, 3 and 1 for step 1 and 2.
Fig. 2. An example of two patches with similar color histograms but different HAH. The central bin histogram is shown in the middle figure and the whole patch histogram is shown in the right figure.

2.4 Phase 4: Practical Usability (Running CBIR on CometCloud)

The CBIR code was ported from Matlab to Java as a native CometCloud application to avoid licensing constrains in non-proprietary resources and to enable the implementation of the application on specialized hardware (e.g., GPU accelerators using JCUDA API or porting to CUDA) in the future. Due to the most computation expensive part is searching query patches within each database image, and final mean shift clustering was implemented within each image of the database, we chose to use Master/Worker programming model, thus each image within the database was assigned to a worker. The implementation using the Master/Worker programming model is shown as Figure 3 (b). A master and a number of workers (one per physical core) form an overlay and synchronize using a tuple space (comet space). The master generates tasks (one for each image or subset of images to be processed) and then the workers pull the tasks and process the associated images simultaneously. In order to improve scalability and fault tolerance, workers store intermediate results on disk rather than returning the results back to the master using the comet space. When the workers finish, the intermediate results are consolidated (which represents a small part of the overall execution). Although the three stages of the algorithm can run in parallel by different workers (e.g., using a pipeline model), we process images sequentially in each worker since the amount of independent tasks involved in a single experiment is extremely large and massive parallelism cannot be assured otherwise.

In order to accelerate the execution of CBIR by exploiting larger scale parallelism, we also used CometCloud to federate a cluster at Rutgers (a Dell Power Edge system with 256 cores in 8-core nodes) with distributed cyber-infrastructure from NSF Extreme Science and Engineering Discovery Environment (XSEDE), NSF FutureGrid, the National Energy Research Scientific Computing Center (NERSC) and public Clouds (Amazon EC2) as shown in Figure 3 (a). Specifically, we used Ranger (Sun constellation with 62,976 cores in 16-core nodes) and Lonestar (with 22,656 cores in 12-core nodes) from XSEDE, Hotel (an IBM iDataPlex system with 672 cores in 8-core nodes) from FutureGrid, Hopper (a Cray XE6 system with 153,216 cores in 24-core nodes) from NERSC, and medium instances from Amazon EC2. The former resources were used through a startup award and the later in pay-as-you-go basis. The different resources are federated using agents: an agent acts as gateway or access point to a system (in login nodes) and a central one orchestrates the rest of agents. The main goal of the federated system is minimizing completion time so we focus on computation. Since in the current implementation computation is prefixed rather than


on-demand and the set of images is fixed, the application assumes that images are already in the computational systems. The image transfer does not impact completion time significantly since most of the systems have a share file system therefore image transfer is required only once, and queuing time is longer than data transfer time. Specifically, transferring the whole set of images over the Internet takes approximately 4 minutes, a chunk of 100 images less than 30 seconds, and a single image only 1-2 seconds.

![Diagram of CometCloud architecture](image)

Fig. 3(a): Overall federation architecture, (b): Master-Worker framework in CometCloud.

3. Results

Figure 4 shows an example of images taken under 10x, 20x and 40x magnification objectives. Searching regions are located from the center of those regions of interest with areas of four times by original query images. Here green crosses are centers of white cells.

![Magnification images](image)

Fig. 4. An example of centers of all white blood cells taken under 10x and 20x magnification objectives.

Figures 5 (a) (b) (c) show the CBIR results under 10x, 20x and 40x magnification objectives, respectively. Here \( P = 90\% \) (the percentage of overlapping of each patch as same size of query image) and \( R = 10 \) (outside 9 rings besides one central rectangular region). The retrieved ROIs were ranked based on the similarity measures to the query ROI. From Figure 5, under 10x, 20x, and 40x, using the most conservative searching \( (P = 90\% \ & \ R = 10) \), query ROI were all accurately retrieved from database. Figure 6 shows our CBIR performances of different types of white blood cells including basophil, eosinophil, lymphocyte, macrophage, monocyte and neutrophil (at 20x objective). All the different types of white cells can be correctly retrieved from a database containing 96 images.
Fig. 5(a). CBIR results of three examples of images from a database under 10x magnification objective. Green rectangular regions are the top ranked retrieval patches similar to query ROI.

Fig. 5(b). CBIR results of three examples of images from databases under 20x magnification objective. Green rectangular regions are the top ranked retrieval patches similar to query ROI.
Fig. 5(c). CBIR results of three examples of images from databases under 40x magnification objective. Green rectangular regions are the top ranked retrieval patches similar to query ROI.

Fig. 6. CBIR results of various types of white blood cells including basophil, eosinophil, lymphocyte, macrophage, monocyte and neutrophil (at 20x). Green rectangular regions are the top ranked retrieval patches which are exactly the same type of the cell in the query ROI.

Under various magnification (M) (10x, 20x and 40x) objectives, the retrieval performances were evaluated by changing the overlapping percentage (P) (from 90% to 50%) and number of outside rings (R) (from 1 to 9). Under 20x and 40x magnification objectives, from 1 ring to 9 rings with from 50% overlapping to 90% overlapping, the query images were correctly retrieved as the most similar patches from the database.
Under 10x magnification objective, when overlapping percentage lower than 70% with number of rings less than 5, the query images were not retrieved as the first ranked retrieved patch, but the second ranked one.

Figure 7 (a) shows the completion time of CBIR algorithm over the database of 925 images for the 50 different configurations (in minutes, using logarithmic scale) and Figure 7 (b) shows the throughput (processed images per minute) that a single node of each of the different platforms can achieve. Each node processes a subset of 100 images and jobs on HPC resources request a single node, priority queues (for short executions) and quite accurate requested CPU time.

Completion time for the federated scenario was obtained with real executions while for sequential and local cluster scenarios completion time is an estimation (due to the limitations of very long executions) based on the actual execution of the subset of configurations. The throughput (with queue) is computed with the observed execution time and number of processed images and the throughput without queuing time is computed subtracting the queuing time from the actual completion time.

Estimation without queuing times is also provided (for federated scenario and HPC resources) to represent the impact of being able to provision elastically HPC resources like Clouds. The results show that CBIR is dramatically speeded up when using the (dedicated) dell cluster at Rutgers with respect to using a single node (from 2.3 weeks of computation to 12 hours). However, using federated infrastructure (i.e., much more resources but not under own control) provides much shorter completion time (about 170 minutes).

The results also show that CBIR could be run in about 58% shorter time if the queuing time in HPC systems was insignificant. It represents the typical tradeoff between the size/length of the job (number of requested cores and execution time) and queuing time (i.e., in general, the longer requested execution time or higher core count the longer queuing time) in batch systems. In our experiments the jobs used a single node (i.e., up to 24 cores) to run a set of 100 images, however, if we used a smaller set of images per job the penalty due to the queuing times would be higher. In case of Amazon EC2 a job processes a smaller set of images because the nodes have smaller core count.

We want to emphasize that the image analysis time varies depending on the particular image, the configuration (e.g., 90% overlapping takes longer) and the platform. Therefore, throughput also varies depending on the platform as shown in Figure 7 (b). As completion time, queuing time impacts throughput, which could be taken into account to implement provisioning strategies and autonomies. However, the results presented in this paper only considers minimizing completion time, regardless of the spend amount of awarded allocation units in National cyber-infrastructure.

The results clearly point out some challenges that should be addressed to effectively run CBIR in HPC resources such as accurately estimating job execution time in such way that jobs can be scheduled rapidly at the same time that jobs are not killed for lasting longer than the requested. Other strategies such as processing smaller set of images per node would increase the parallelism level but it would require provisioning HPC resources more elastically and on-demand (as Cloud resources), which is in our future research agenda.

![Fig. 7(a). Completion time, (b). throughput using different configurations and platforms](image-url)
4. Conclusion and Future work

Within the paper, we demonstrated a newly developed CBIR system and demonstrate its application on digitized peripheral blood smears using three different image datasets. Using center identification of white blood cells before searching on image database, it significantly reduces the computational time by avoiding unrelated regions. Meanwhile hierarchical searching significant reduces the searching time and speeds up the CBIR process. Using the most conservative searching (90% overlapping with 9 rings outside the central region), all the query images under 10x, 20x and 40x magnification objectives had been correctly retrieved from the image databases. Various types of white blood cells including basophil, eosinophil, lymphocyte, macrophage, monocyte and neutrophil (at 20x) can be correctly retrieved efficiently. To improve the discriminative power, high-level local features can be included and tested. Besides blood smear, this framework can be extended to other medical and/or clinical applications in the future.

CometCloud, combined with federated HPC cyber-infrastructure(Grids and Clouds) are utilized in this project. The results demonstrated that CBIR execution could be dramatically sped up, from weeks to minutes. The proposed federated system presents many opportunities and challenges. We plan to address how to elastically provision HPC resources on-demand, and how to exploit heterogeneous resources from the point of view of their capabilities (e.g., using the Matlab incarnation of CBIR when licenses are available or the GPU incarnation when resources with accelerators are available.

Acknowledgements: This research was funded, in part, by grants from the National Institutes of Health through contract 5R01CA156386-06 and 1R01CA161375-01A1 from the National Cancer Institute; and contracts 5R01LM009239-04 and 1R01LM011119-01 from the National Library of Medicine. Additional support was provided by a gift from the IBM International Foundation. This work is sponsored by the National Science Foundation (NSF) Office of Industrial Innovation and Partnerships Industry/University Cooperative Research Center (IUCRC) Program under award 0758596. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by NSF grant number OCI-1053575, used resources of FutureGrid, which is supported in part by the NSF under Grant No. 0910812, and used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. CometCloud was developed as part of the NSF Cloud and Autonomic Computing Center at Rutgers University. The project is also partially supported by the National Center for Research Resources and the National Center for Advancing Translational Sciences, National Institutes of Health, through Grant UL1TR000117 (or TL1 TR000115 or KL2 TR000116). The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.

References
Content-based Parallel Sub-image Retrieval

Fuyong Xing\textsuperscript{(1,2)}, Xin Qi\textsuperscript{(3)}, David J. Foran\textsuperscript{(3)}, Tahsin Kurc\textsuperscript{(4)}, Joel Saltz\textsuperscript{(4)}, Lin Yang\textsuperscript{(1,2)}

\textsuperscript{1} Division of Biomedical Informatics, Dept. of Biostatistics, University of Kentucky
\textsuperscript{2} Dept. of Computer Science, University of Kentucky
\textsuperscript{3} Center for Biomedical Imaging & Informatics, The Cancer Institute of New Jersey
\textsuperscript{4} Center for Comprehensive Informatics, Emory University

Abstract. In this work, we propose a three stage-based content-based sub-image retrieval (CBSIR) framework for whole slide tissue and tissue microarray images. The user selects an image patch (sub-image) as the query, and the retrieval system returns a list of images containing similar patches. It first performs a hierarchical searching based on a newly developed feature named hierarchical annular histogram, which is scale and rotation invariant and designed to capture the structure information of image patches. Hierarchical searching iteratively discards a certain percentage of less similar candidates and the final result will be refined by computing a color histogram from 8-equally-divided segments in each square annular bin in the second stage. To prevent similar candidates in one image from densely overlapping with each other, mean-shift clustering is applied to generate the final retrieved image patches. This hierarchical searching schema is fast and significantly speeds up the subsequent refined process. To further decrease the execution time, we employ parallel processing. Since the task of searching for similar image patches can be done independently, we adopt a demand-driven master-worker implementation. With this approach, the query patch is broadcast to all worker processes. Each worker process is dynamically assigned an image by the master process to search for and return the list of similar patches in the image. We also use the hierarchical search algorithm to reduce the list of images to be processed when resources are limited and can benefit from sharing among other clients and processes.

1 Introduction

The exponential growth of images and video in last decade has resulted in an increasing need for efficient content-based image retrieval (CBIR), which can detect and locate similar images in large-scale collections given a query. For medical diagnoses assistance, several state-of-the-art CBIR systems \cite{1,2,3,4} have been designed to support the processing of queries across separate images. However, it is not unusual that users may be interested in subregion searching (usually an image patch exhibiting specific patterns or structures, containing an object). Given this subregion, the system should be able to return other patches within the same images which contain the localized subregions exhibiting similar features. This process is called content-based sub-image retrieval (CBSIR). One
of the advantages of CBSIR is that the relevance of images is less effected by changes in image viewpoint or background clutter [5]. In practice, this approach makes it possible for a pathologist to select an area or object of interest within a digitized biospecimen as a query to reliably search corresponding regions in either the same biospecimen or within cohorts of patients having the same disease and make informed decision regarding the treatment regimens based upon the comparison.

Recently researchers have proposed many state-of-the-art methods to perform CBSIR related to both natural and medical images. Luo and Nascimento [6] introduced relevance feedback by applying a tile re-weighting approach to assign penalties to tiles that compose database images and update the penalties for all retrieved images within each iteration. This procedure is time consuming due to the feedback learning. To perform region-of-interest (ROI) query, Vu et al. [7] presented a SamMatch framework-based similarity model, which is scale invariant. Meanwhile, an R*-tree based indexing technique is employed to obtain faster retrieval. A hashtable-based method for image object retrieval is presented in [8], which applied intra-expansion and inter-expansion strategies to boost the hash-based search quality using bags of features to present images. A part-based approach reported in [9] to solve subimage retrieval problem by synthesizing DoG detector, PCA-SIFT, and local sensitive hashing searching algorithm. However, its time cost is relatively high because of the large amount of features need to be computed.

To perform a large-scale subregion retrieval, the method reported in [10] employed approximate K-means and hierarchical K-means to build large vocabularies and introduced a randomized tree-based quantization algorithm. Furthermore, a spatial verification stage is used to re-rank the results returned from the bag-of-words model by estimating a transformation between the query region and each target image. Tang et al. [11] incorporated a contextual synonym dictionary to the bag-of-feature framework for large scale visual object search, where synonym words are used to describe visual objects with the same semantic meaning and are identified via measuring the similarities of their contextual distribution. This method is simple and cheap, and proven to be effective extensively. A fast and efficient sub-window search (ESS) algorithm is presented in [12] to localize interest regions (objects) using branch-and-bound scheme which allows efficient maximization of a large class of classifier functions over all possible sub-images. ESS is guaranteed to retain global optimal solution in its search and the speed advantage allows the use of more complex and better local classifier such that it can give the state-of-the-art performance. Based on ESS, Lampert [5] introduced a new box set parametrization that is suitable for subregion retrieval and a two layer branch-and-bound scheme to localize objects in large image collections. Another subregion driven image retrieval method can be found in [13], which represented objects with a set of viewpoint invariant region descriptors and used spatial layout to rank the retrieved regions.

For subregion retrieval in medical image dataset, a structured visual search method is presented in [14]. It first classified images into predefined classes such
that region of interest (ROI) correspondences are only considered between images of the same class. Next, it computed approximate global registration for images within that class. Finally, the ROI is online refined in a target image using the previous registration as a guide. This framework is generic and can be extended to other modalities. Thoma et al. [15] proposed a method for ROI query in CT scans. It first employed instance-based regression in combination with interpolation techniques for mapping the scan slides to height model of human body. Next, a query algorithm is designed to find a stable mapping while deriving a minimal amount of matching points.

In this work, we propose a three stage-based CBSIR framework for whole slide scanned tissue microarray images. The flow chart is shown in Figure 1. The algorithm first performs hierarchical searching with a newly developed feature called a hierarchical annular histogram (HAH), and next refines searching by computing color histogram from 8-equally-divided segments of each squarely annular bin, to which we referred as refined HAH. Finally, mean-shift is employed to cluster densely overlapping candidates to generate final retrieved image patches. A master-worker style parallel execution strategy is employed to reduce execution time on parallel machines. This strategy uses a demand-driven assignment scheme in which images are assigned by a master process to worker processes dynamically when worker processes become idle. This parallelization strategy is suitable for this application since search for images patches similar to the query patch can be performed independently. The hierarchical searching can also be used to reduce the list of images to be processed and returned to the client application. This approach can decrease resource usage when resources are limited and shared by other clients and applications.

The rest of this paper is organized as follows: section 2 introduces the proposed HAH feature and three stage-based CBSIR framework. Experimental results are presented in section 3 and section 4 concludes the paper.

2 Sub-image Retrieval

For a CBSIR system, users usually need to select an image patch that specifies an object or represents a certain pattern on the query image (Figure 2(a)), and this
patch is used as the query patch to retrieve images containing similar patches from the database. This process is also known as object-based image retrieval. To rank the candidates, features need to be extracted to describe the image patches, and SIFT descriptor is one of the most popular local features used in recent literatures [10,12]. However, it is difficult and time consuming to directly extend SIFT feature to medical image description. In this work, we propose the hierarchical annular histogram (HAH), which is described in next sub-section.

### 2.1 Hierarchical Annular Histogram (HAH)

For a given image patch, we first segment it into several closed bins with equal intervals, shown in Figure 2(b). Next, color histogram of each bin is computed and then all the histograms are concatenated to form a single histogram, which we called hierarchical annular histogram (HAH). We can obtain the following benefits by using this feature: (1) It is scale and rotation invariant. (2) It captures spatial configuration of image local features. and (3) It is suitable for hierarchical searching in the following parallel sub-image retrieval.

With HAH, the discriminative power of each image patch descriptor has been significantly improved compared with traditional color histogram (TCH). For medical images, it is very likely that image patches with different structures have quite similar intensity distribution as a whole, but different HAH instead.

### 2.2 Three Stage-based CBSIR Framework

The proposed CBSIR framework consists of three stages: hierarchical searching, refined searching, and mean-shift clustering. The hierarchical searching scheme is an iterative process that discards less similar candidates within each iteration. It begins with calculating the color histograms of the inner (first) central bins for candidate patches and compares them with those of the query patch. Based on the defined dissimilarity, it will remove a certain percentage of candidates after the first iteration. For the second iteration, it only calculates the color histograms from the second central bin and further delete a certain percentage of candidates.
by computing the dissimilarity with the query patch’s histograms from the two inner bins. This process is operated iteratively, and the final candidates which passes all these steps will be the image patches most similar to the query patch. These final results will be further refined by computing color histogram from 8-equally-divided segments of each square annular bin. To rank the candidates in each step, we simply define the dissimilarity $D(H(X_q), H(X_r))$ between query $X_q$ and candidate $X_r$ patches as

$$D(H(X_q), H(X_r)) = ||H(X_q) - H(X_r)||_2 \quad (1)$$

where $H(X)$ is the color histogram of patch $X$. A smaller $D$ demonstrates strong similarity between the candidate and the query patch.

The hierarchical searching procedure can greatly reduce the time complexity, because it only computes one bin of HAH and rejects a large portion of candidates within each iteration. The number of candidates moving to the next step is largely reduced by rejecting the obvious negative candidates. In Figure 3, we show the whole procedure for the hierarchical searching procedure.

In the refined stage, each annular bin has been equally divided into 8 segments (Figure 2(c)), and the color histogram in each segment is computed and composed together to generate one single histogram. The final candidates are chosen based on the dissimilarity defined in (1). Due to the very limited number of candidates passing the hierarchical searching stage, this refined process is not particularly time consuming. Mean-shift is applied to finally refine the searching results.

### 2.3 Parallel Execution

The hierarchical CBSIR will reduce the cost of searching for images patches similar to a given query patch compared to a non-hierarchical algorithm. Nevertheless, processing a large number of high resolution images may take prohibitively long on a workstation, limiting the use of CBSIR. We have developed a parallel implementation to address this issue. The parallel implementation employs a
master-worker parallelization strategy. The objective of our implementation is to provide a high-throughput processing version in which datasets with large number of images can be processed quickly. Our approach is based on the fact that the similarity computation of an image patch can be performed independently of other patches in different images. Thus, multiple images and image tiles can be processed concurrently in parallel.

In our implementation, images and image tiles form the basic unit of processing. If an image is partitioned into multiple disjoint tiles, each tile needs to be padded in $x$- and $y$-dimensions by an amount equal to the $x$-resolution and $y$-resolution of the maximum query patch, respectively. This is required to guarantee that no patches matching the query patch are divided across tile boundaries.

The master-worker implementation uses a demand-driven strategy for assignment of images (or image tiles) to processors. Each worker process requests work when it has finished the previous task and becomes idle. A master process then assigns an image to the worker process. Since hierarchical searching may eliminate some image patches from further consideration, the cost of processing each image will be different. This could create load imbalance across worker processes if a static assignment is used.

Resources used to provide CBSIR services may be constrained and shared across multiple applications. On a typical parallel system, backend computation resources will be accessed by multiple applications. Moreover, if the parallel implementation is deployed as a remote service, multiple clients might submit requests. The service then would have to share resources across multiple client requests. In these cases, it would be beneficial to control and decrease resource usage and avoid long execution times against large image datasets. The hierarchical search stage can be utilized for this purpose.

Instead of processing all images or image tiles through all the stages of the CBSIR workflow in Figure 1, a pre-determined number of iterations of hierarchical searching can be applied to each image - the number of iterations could be provided by the client or set in the system as default. At the end of this step, a similarity measure can be computed and assigned to each image (or image tile). During each iteration of hierarchical searching, each image patch is assigned a similarity value, indicating how similar the image patch is to the query patch. These values can be used to compute a similarity measure for a given image. For example, a similarity measure for an image could be the number of image patches whose similarity values exceed a user-defined threshold. Alternately, the average of similarity values of all the image patches in an image could also be used as a similarity measure for the image. Once a similarity measure is computed for each image in the dataset, the images are sorted based on similarity values and the client chooses a subset of images for further processing through the CBSIR workflow. We have prototyped this approach in our parallel implementation to evaluate its impact on execution times of a client request.
3 Results

In this section, we present performance results from our parallel implementation of the CBSIR framework. The experiments were carried out on a distributed memory computation cluster. Each node of the cluster has four 6-core CPUs and are connected to each other via an Infiniband switch. We used a dataset with 96 images and a single query patch in the experiments.

Figure 4 shows the execution time of hierarchical and non-hierarchical CBSIR for processing 96 images using different numbers of CPU cores. In this experiment, each image is a unit of task, and the number of CPU cores varies from 8 to 64 on 8 computation nodes. The non-hierarchical CBSIR processes each image by scanning all image patches and computing similarity values for each patch, unlike the hierarchical CBSIR which eliminates some of the image patches from further processing. As illustrated in the figure, the hierarchical algorithm takes much less time than the non-hierarchical version. These results show that one can achieve substantial performance benefits by using the hierarchical CBSIR. The execution time decreases for both algorithms as more cores are used, as expected. Our results indicate that parallel processing can be efficiently employed to decrease processing times and make the processing of large datasets feasible.

The first set of experiments also shows that even when hierarchical CBSIR is executed on a parallel machines, the execution time for processing a large dataset may be high - it took about 2.3 hours to process 96 images on 64 CPU cores. In the next set of experiments, we look at the use of the hierarchical searching step to reduce the number of images to be processed, as is described in Section 2.3. In these experiments we used 48 images and 16 cores (on 8 computation nodes). We first executed the hierarchical searching step on all the images, and then selected 16 images based on the similarity measures. The selected images are then processed using the full hierarchical CBSIR algorithm. Figure 5 shows the execution
times of the hierarchical searching step with different number of iterations (the first three columns - one iteration, two iterations, three iterations – in the figure) as well as processing 16 and 48 images on 16 cores. As is seen from the figure, the cost of the hierarchical searching steps increases as the number of iterations executed in that step increases, as expected. However, the cost of this step is still considerably small compared to processing all the images. The column (three + 16) shows the execution time of processing 16 images plus the cost of the hierarchical searching step with three iterations. When the hierarchical searching step is used to select a smaller subset of images for processing, the execution time can be reduced considerably: approximately 5200 seconds for processing 16 images including the hierarchical searching step with three iterations vs 11800 seconds for processing 48 images.

4 Conclusion

In this paper, we have presented the design and implementation of a content-based sub-image retrieval (CBSIR) framework. The framework employs a hierarchical searching step to reduce the cost of extracting images patches from a high-resolution microscopy image that are similar to a query patch. We also presented a prototype implementation of the framework on a parallel machine. Our results show that performance savings can be significant with the hierarchical CBSIR compared to non-hierarchical CBSIR. In addition, the hierarchical searching step can be used to reduce the number of images to be analyzed using...
a user-defined similarity measure. The cost of the hierarchical searching step is small enough so that substantial reduction in resource usage can be achieved when a subset of images are selected and processed, even when the cost of the hierarchical searching step is added to the overall execution time.

5 Acknowledgement

This research was funded, in part, by grants from the National Institutes of Health through contracts 5R01CA156386-06, 1R01CA161375-01A1 from the National Cancer Institute; contract HHSN261200800001E by the National Cancer Institute; and contracts 5R01LM009239-04 and 1R01LM011119-01 from the National Library of Medicine, R24HL085343 from the National Heart Lung and Blood Institute, NIH NIBIB BISTI P20EB000591, RC4MD005964 from National Institutes of Health, and PHS Grant UL1TR000454 from the Clinical and Translational Science Award Program, National Institutes of Health, National Center for Advancing Translational Sciences. Additional support was provided by a gift from the IBM International Foundation. The project is also partially supported by the National Center for Research Resources and the National Center for Advancing Translational Sciences, National Institutes of Health, through Grant UL1TR000117 (or TL1 TR000115 or KL2 TR000116). The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.

References

A MapReduce Approach for Ridge Regression in Neuroimaging-Genetic Studies

Benoit Da Mota\textsuperscript{1,3}, Michael Eickenberg\textsuperscript{1}, Soizic Laguitton\textsuperscript{2}, Vincent Frouin\textsuperscript{2}, Gaël Varoquaux\textsuperscript{1}, Jean-Baptiste Poline\textsuperscript{2}, and Bertrand Thirion\textsuperscript{1}

\textsuperscript{1} Parietal Team, INRIA Saclay, France
\textsuperscript{2} CEA, DSV, I2BM, Neurospin, France
\textsuperscript{3} Parietal Team, MSR-INRIA joint centre, France
{benoit.da_mota,michael.eickenberg, gael.varoquaux,bertrand.thirion}@inria.fr
{soizic.laguitton,jbpoline}@gmail.com
{vincent.frouin}@cea.fr

Abstract. In order to understand the large between-subject variability observed in brain organization and assess factor risks of brain diseases, massive efforts have been made in the last few years to acquire high-dimensional neuroimaging and genetic data on large cohorts of subjects. The statistical analysis of such high-dimensional and complex data is carried out with increasingly sophisticated techniques and represents a great computational challenge. To be fully exploited, the concurrent increase of computational power then requires designing new parallel algorithms. The MapReduce framework coupled with efficient algorithms permits to deliver a scalable analysis tool that deals with high-dimensional data and hundreds of permutations in a few hours. On a real functional MRI dataset, this tool shows promising results.

Keywords: Bio-statistics, Neuroimaging, Genetics, Ridge regression, Permutation Tests.

1 Introduction

Using genetics information in conjunction with brain imaging data is expected to significantly improve our understanding of both normal and pathological variability of brain organization. It should lead to the development of biomarkers and in the future personalized medicine. Among other important steps, this endeavor requires the development of adapted statistical methods to detect significant associations between the highly heterogeneous variables provided by genotyping and brain imaging, and the development of the software components that will permit large-scale computation to be done.

In current settings, neuroimaging-genetic datasets consist of a set of \textit{i)} genotyping measurements at given genetic loci, such as Single Nucleotide Polymorphisms (SNPs) that represent a large amount of the genetic between-subject variability, on the one hand, and \textit{ii)} quantitative measurements at given locations (voxels) in three-dimensional images, that represent e.g. either the amount
of functional activation in response to a certain task or an anatomical feature, such as the density of grey matter in the corresponding brain region.

Most of the efforts so far have been focused on designing association models, and the computational procedures used to run these models on actual architectures have not been considered carefully. For instance, permutation tests of simple linear association models have been deemed as inefficient in some of these studies, e.g. [11]; however, they can be replaced by analytical tests only in very specific cases and under restrictive assumptions. Gains in sensitivity might be provided by multivariate models in which the joint variability of several genetic variables is considered simultaneously. Such models are thought to be more powerful [13, 1, 5, 7], because they can express more complex relationships than simple pairwise association models. The cost of unitary fit is high due to high-dimensional, potentially non-smooth optimization problems and various cross-validation loops needed to optimize the parameters; moreover, permutation testing is necessary to assess the statistical significance of the results of such procedures in the absence of analytical tests. Multivariate statistical methods require thus many efforts to be tractable in this problem on both the algorithmic and implementation side, including the design of adapted dimension reduction schemes. In this work we will consider the simplest approach, ridge regression [5], that is powerful for detecting multivariate associations between large variable sets, but does not enforce sparsity in the solution.

Working in a distributed context is necessary to deal with the memory and computational loads, and yields specific optimization strategies. Once the unitary fit cost has been minimized, the main task when implementing such natural data parallel applications is to choose how to split the problem into smaller sub-problems to minimize computation, memory consumption and communication overhead. For the first time, we propose an efficient framework that can manage ridge regression with numerous phenotypes and permutations.

In Section 2, we present our sequential algorithm, then we describe our framework to distribute efficiently the computation on large infrastructures. Experimental results on simulated and real data are presented in Section 3.

2 Methods: the computational framework

Ridge regression of neuroimaging genetics data is clearly an embarrassingly parallel problem, which can be easily split into smaller tasks. Our computational framework relies on an adapted workflow summarized in Fig. 1, in which sub-tasks are optimized for the sake of efficiency. To simplify the presentation we first describe the core algorithm and then the workflow.

2.1 Optimizing the ridge regression algorithm

The map step, i.e. the scoring of ridge classifiers, is the most demanding in computation time (< 99.9% in our final implementation) and thus has to be optimized in priority. The computational burden mostly depends on the ridge
Ridge Regression for Neuroimaging-Genetic Studies

Our algorithm performs Ridge Regression for multiple targets and multiple individual penalty values. It solves the following problem:

\[ \hat{\beta}_{ij} = \arg\min_{\beta} \| y_i - X\beta \|_2^2 + \lambda_{ij} \| \beta \|_2^2, \ i \in [1, p], j \in [1, J] \]

where \( X \in \mathbb{R}^{n \times p} \) is the gene data matrix, \( y_i \in \mathbb{R}^n \) is a variable extracted from brain images, \( \hat{\beta}_{ij} \in \mathbb{R}^p \) is the estimated coefficient vector, and \( \lambda_{ij} > 0 \) is the penalty term where \( j \) indexes \( J \) different penalties for the target \( y_i \). We obtain the solution using the singular value decomposition (SVD) of \( X \), which we write \( X = USV^T \), truncated to non-zero singular values. In the full rank case and for \( p > n \) we have \( U \in \mathbb{R}^{n \times n} \) and \( V^T \in \mathbb{R}^{n \times p} \), while \( S \) is a diagonal matrix with entries \( s_k \), \( 1 \leq k \leq n \). For one \( \hat{\beta}_{ij} \) we have

\[ \hat{\beta}_{ij} = V \text{diag} \left( \frac{s_k}{s_k^2 + \lambda_{ij}} \right) U^T y_i \]

All \( \hat{\beta}_{ij} \) are calculated with the same SVD, it is reused (and cached). For all \( i \), \( U^T y_i \) is pre-calculated, which is conveniently and effectively done by multiplying the matrices \( U^T \) and \( Y \) where the columns of \( Y \) are the \( y_i \). Since for a given \( j \) every target \( i \) potentially has a different penalty associated, the shrinkage operation \( \frac{s_k}{s_k^2 + \lambda_{ij}} \) is not writable as a matrix multiplication against \( U^T Y \). However, it is a linear operation on matrices, and by defining \( \Sigma \in \mathbb{R}^{n \times p} \) with \( \Sigma_{ki} = \frac{s_k}{s_k^2 + \lambda_{ij}} \) for a fixed \( j \), it can be written as the pointwise matrix product

\[ \hat{\beta} = V (\Sigma \circ U^T Y) \]

These are the operations implemented by our algorithm for the \( J \) different sets of penalties, using \( J \) different matrices \( \Sigma \). With a pre-calculated SVD and \( U^T Y \), the cost of this operation is \( \mathcal{O}(npN) \), where \( N \) is the number of target variables.

Care was taken of computational/hardware sources of optimization, like CPU cache issues. For instance, matrix-based operations are used instead of vector-based operations to optimize the use of advanced vector extensions instructions set in new CPU. Our Python code uses the Numpy/Scipy/Scikit-learn scientific libraries, which rely on standard and optimized linear algebra libraries (Atlas or MKL) that are several order of magnitude faster than naive code. Next, we need to consider evaluation and parameter setting procedures:

- the power of the procedure is measured by the ratio of explained variance, computed within a shuffle-split loop that leaves 20\% of the data as a test set at each of the ten iterations;
- to select the optimal shrinkage parameters, \( J = 5 \) values are tested first, then a grid refinement is performed where five other parameters are tested;
- each shrinkage parameter of the ridge regression is evaluated using an inner 5-folds cross validation loop.

This setting thus needs approximately 500 ridge regressions for one phenotype and one permutation.
2.2 The distributed algorithm

The MapReduce framework [2,3] seems the most natural approach to handle this problem and can easily harness large grids. The Map step yields explained variance for an image phenotype and for each permutation, while the reduce step consists in collecting all results to compute statistic distribution and corrected p-values. Sub-tasks are created in a way that minimizes inputs/outputs (I/O). By essence, permutations imply computations on the same data after shuffling. The permutation procedure is thus embedded in the mapper, so that all permutations loops are run on the same node for a given dataset and the problem is split in the direction of the brain data. Figure 1 gives an overview of our framework.

Shared cache is a crucial feature since the Map step is dominated by costly SVDs. For instance, with 1,000 phenotypes and 1,000 permutations, each SVD in the inner CV loop is required 2 millions times and costs few tens of second. A shared cache on NFS, provided by the Joblib Python library [12], coupled with system cache saves many computations.

3 Results

We present three types of results. First, we present the performances of our distributed framework. Then, we illustrate the interest of our approach on simulated data with known ground truth. Finally, we present results on a real dataset.

3.1 Performance evaluation of the procedure

To illustrate the scalability of our Map-Reduce procedure, we execute the whole framework on a cluster of 20 nodes; each one is a 2 × Intel(R) Xeon(R) CPU X5650 (6 cores) @ 2.67GHz with 48GB of memory, connected with Gigabit Ethernet LAN; all files were written on the NFS storage file-system; our mapper
runs with the Enthought Python Distribution (EPD 7.2-2-rh5 64 bits) with scikits-learn 0.11 [8] and with the MKL as linear algebra library with OpenMP parallelization disabled; the workflow is described and submitted with the soma-workflow software [6]. This framework makes it possible

i) to describe a set of independent tasks that are executed following an execution graph and

ii) to execute the code by submitting the graph to classical queuing systems operating on the cluster. We report in Fig. 2 the result of an execution with almost all the 240 cores available during all the run. The workflow is composed by 1,000 mappers and 1 reducer tasks. The mappers represent 99.9% of the total of serial computation time. Once the SVD are cached, the execution time of a map task is around 20 minutes. We can see in Fig. 2 that after 88 minutes, we use only few cores, but all the unused cores are available for other users. This comes from the number of tasks: on 240 cores, after 4 batches of 240 tasks, only 40 are left. To improve the global speedup, we could split the problem into smaller pieces to decrease the time of the mappers or we could choose a more optimal splitting. We have not explored these possibilities yet.

3.2 Simulated Data

We simulate functional Magnetic Resonance Images (fMRI) from real genetic data obtained from the Imagen database [10]. We use the number of minor alleles for each SNP and we assume an additive genetic model. We use only the first chromosome in which ten random SNPs produce an effect in a spherical brain region, centered at a random position in the standard space, then intersected with the support of grey matter using a mask computed for the Imagen dataset (see below). We add i.i.d. Gaussian noise, smoothed spatially with a Gaussian kernel ($\sigma = 3mm$), to model other variability sources. The effect size and the Signal-to-Noise Ratio (SNR) can vary across simulations. Then 1,000 imaging phenotypes are obtained by computing the mean signal in brain parcels that were created using a Ward Agglomeration clustering.

To assess our approach, ten different datasets were generated and were run on our framework with $P=200$ permutations to estimate the distribution of the maximum explained variance under the null hypothesis. Results are given in Table 1 and show that our method detects 8 effects among 10 simulations with a p-value $p < .05$. The results do not give evidence of the influence of the SNR simulation nor of the volume of the effect on the test sensitivity.
3.3 Results on a real dataset

We used data from Imagen, a large multi-centric and multi-modal neuroimaging database [10] containing functional magnetic resonance images (fMRI) associated with 99 different contrast images in more than 1,500 subjects. The dataset is built on the first batch of subjects of the study. Regarding the fMRI data, the protocol in [9] was used, which yields the [angry faces - neutral] functional contrast (i.e. the difference between watching angry faces or neutral faces).

**Imaging phenotype.** Standard preprocessing, including slice timing correction, spike and motion correction, temporal detrending (functional data), and spatial normalization (anatomical and functional data), were performed using the SPM8 software and its default parameters; functional images were resampled at 3mm resolution. Obvious outliers detected using simple rules such as large registration or segmentation errors or very large motion parameters were removed after this step. The [angry faces - neutral] contrast was obtained using a standard linear model, based on the convolution of the time course of the experimental conditions with the canonical hemodynamic response function, together with standard high-pass filtering procedure and temporally auto-regressive noise model. The estimation of the model parameters was carried out using the SPM8 software. A mask of the grey matter was built by averaging and thresholding the individual grey matter probability maps. Subjects with too many missing data (imaging or genetic) or not marked as good in the quality check were discarded. An outliers detection [4] was run and 10% of the most outlier subjects were eliminated.

**Genotype.** We keep only SNPs in the first chromosome with less than 2% missing data. All the remaining missing data were replaced by the median over the subjects for the corresponding variable. The age, the sex and the acquisition center were taken as confounding variables.

The final dataset contains 1,229 subjects, 1,000 brain parcels, 31,790 SNPs and 10 confounding variables. Our Map-Reduce framework was run with P=1,000 permutations to assess statistical significance. The workflow takes approximately

<table>
<thead>
<tr>
<th>#</th>
<th>Simul. #</th>
<th>Volume (mm³)</th>
<th>Average SNR</th>
<th>Best Parcel explained variance</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3375</td>
<td>0.19</td>
<td>0.022</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3348</td>
<td>0.66</td>
<td>0.042</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2457</td>
<td>0.30</td>
<td>0.056</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2754</td>
<td>0.54</td>
<td>0.033</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3213</td>
<td>0.22</td>
<td>0.007</td>
<td>0.35</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3348</td>
<td>1.50</td>
<td>0.027</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1431</td>
<td>0.55</td>
<td>0.031</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1890</td>
<td>0.66</td>
<td>0.005</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3375</td>
<td>0.19</td>
<td>0.036</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3132</td>
<td>0.41</td>
<td>0.026</td>
<td>0.005</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Results on the simulated datasets p-value the p-value corresponding to the given ratio of explained variance, obtained by 200 permutations)
Fig. 3. Location of the brain parcel with a significant explained variance ratio (exp. var. = 0.019, \( p \approx 0.048 \), corrected for multiple comparisons) on the real dataset.

9 hours on the previously described 240 cores cluster, for a theoretical serial time around 75 days (i.e. a speed-up of approximately 200). Only one parcel is detected with a corrected p-value \( \leq 0.05 \). A view of the location of the detected parcel is reported in Fig. 3.

4 Conclusion

Penalized linear models represent an important step in the detection of associations between brain image phenotypes and genetic data, which faces a dire sensitivity issue. Such approaches require cross validation loops to set the hyperparameters and for performance evaluation. Permutations have to be used to assess the statistical significance of the results, this yielding prohibitively expensive analyses. In this paper, we present an efficient and scalable framework that can deal with such a computational burden and that we used to provide a realistic assessment of the statistical power of our approach on simulations. Our results on simulated data highlight the potential of our method and we provide promising preliminary results on real data, including one multivariate association that reaches significance. To the best of our knowledge, this is the first result of that kind in a brain-wide chromosome-wide association study, although it needs to be reproduced to be considered as meaningful.

Acknowledgment Support of this study was provided by the IMAGEN project, which receives research funding from the European Community’s Sixth Framework Program (LSHM-CT-2007-037286) and coordinated project ADAMS (242-257) as well as the UK-NIHR-Biomedical Research Centre Mental Health, the MRC-Addiction Research Cluster Genomic Biomarkers, and the MRC program grant Developmental pathways into adolescent substance abuse (93558). This research was also supported by the German Ministry of Education and Research (BMBF grant # 01EV0711). This manuscript reflects only the author’s views and the Community is not liable for any use that may be made of the information contained therein. This work was supported in part by the joint INRIA - Microsoft Research Center. The experiments presented in this paper were carried out using the computing facilities of the CATI.
References


