Video++, a Modern Image and Video Processing C++ Framework.

Matthieu Garrigues
ENSTA-ParisTech,
828 Boulevard des Maréchaux,
91762 Palaiseau CEDEX, France,
http://www.ensta-paristech.fr
matthieu.garrigues@ensta-paristech.fr

Antoine Manzanera
ENSTA-ParisTech,
828 Boulevard des Maréchaux,
91762 Palaiseau CEDEX, France,
http://www.ensta-paristech.fr
antoine.manzanera@ensta-paristech.fr

***Abstract*—We present in this paper *Video*++, a new framework targeting image and video applications running on multi-core processors. While offering a high expressive power, we show that it generates code running up to 32 times faster than the naive equivalents. Taking advantage of the new C++11/C++14 features, tools, we propose simple abstractions matching the performance of hand optimized code. This paper gives an overview of the library and demonstrates its efficiency with some benchmarks.**

Introduction

Since the 60s, computer scientists have been using computers to automate the processing of digital images for many fields like medical imaging, satellite imagery, video surveillance, photography and others. Developers of theses applications have always faced a major challenge: How to deal with a very large amount of data in the shortest period of time, with the available computing resources? Today, a $2cm^2$ processor executes 10 000 faster than ENIAC, the biggest computer of year 1946. This enables computers to handle a lot more pixels per second. However, the growing demand of real-time, embedded, high-resolution video applications, and the heterogeneity of the modern processors has led to more and more complexity in processor designs.

To overcome this increasing complexity, the introduction of high level programming languages like C++ provided abstractions allowing programmers to write shorter code, without affecting the performances of the generated binary code.

Computer vision applications need genericity and performance. For example, an image varies depending of the sensor or the storage format. But, the algorithmic principle does not vary much whether it has 8 bits or 32 bits values. It is desirable to write one generic algorithm that handles many types of images, running as fast as the specialized version that one could write for each of them.

For this reason, C++ and its concept of template was widely adopted by the community and many C++ frameworks were created to ease the writing of fast image processing application with generic, multi-dimensional image types and libraries of algorithms.

The first standard of C++ was released in 1998, and the last one in 2011. The next, C++14 is currently a draft but already implemented in major compilers like G++ [1] or Clang++ [2]. The C++11 and C++14 standards greatly improve the expressive power of the language.

We present in the following the existing image processing frameworks, then present our motivations.

The most popular framework is the OpenCV library [3] which was originally written in C, then extended to C++. It contains a comprehensive set of computer vision primitives covering many fields of applications. However, the fact that it was first written in C prevents it from taking full advantage of C++. Olena [4] is a platform embedding Milena, a C++ image processing library. It pushes the limit of genericity by offering tools to write once an algorithm running on a broad range of types of images like 2D and 3D dense images, graphs, run-length encoded images and others. VIGRA [5], CIMG [6] and ITK [7] are C++ toolkits for image processing. Like Olena, they take advantage of C++ to implement generic image types and image processing primitives. Halide [8] focuses on high performance computing. It provides C++ abstractions and a compiler with optimizations specialized for semi-automatic parallelization of image processing kernels.

All these frameworks are mainly written with C++98, lacking powerful features like lambda functions. Furthermore, to workaround the language limitations they often rely on complex and verbose meta programming, or heavy error-prone use of the C++ preprocessor.

We propose in this paper to design a new framework based on the new C++ standards and targeting multi-core architectures. The resulting code is *Video*++, a fast tiny library of 1 500 lines that enables a developer to write algorithms running up to 32 times faster than the naive versions while being as simple to write.

The main idea behind Video++ is to redesign from scratch an image processing framework taking advantage of the new C++ standards, new tools like OpenMP4, and progress in compiler optimization. We believe that these advances can simplify both the writing and the use of such framework, leading to shorter code and faster executables.

By not explicitely using architecture-specific vector instruction such as SSE or NEON, *Video*++ is portable and lets the compiler generate optimized code for the target instruction set. Furthermore, the core of the library takes advantage of C++

templates to generate code that is trivial for the compiler to optimize.

Section I presents the new C++ features on which *Video++* relies. Then, section II gives an overview of the library. And in section III, by benchmarking typical algorithms, we show that, while having a much higher expressive power, the framework matches the performances of a parallel hand-optimized code. Actually, it outperforms naive implementations by a factor up to x32, and OpenCV by a factor up to x5.6. The framework is open source and accessible online via the git repository [9].

I. C++11 AND C++14

C++11 and C++14 standards bring new features giving the opportunity to write image and video processing frameworks that are more generic and more performant. This section presents the subset of these features used by our framework.

A. Lambda Functions (C++11) and Generic Lambda Functions (C++14)

The C++ standard committee introduced with C++11 the concept of lambda function [10]. They refined it in C++14 with the addition of generic argument [11]. It allows to define anonymous functions, that may be treated as immutable variables, and passed as arguments to other functions.

Listing 1. Definition of a lambda function.

```
auto add = [] (int a, int b)
{
  return a + b;
};
```

This construct also enables the definition of closures, letting the programmer access to external variables, by copy with operator = or by reference with operator &:

Listing 2. Lambda closure.

```
int X = 1;
int Y = 2;
auto fun = [&X, =Y] (int a, int b)
{
  X = 4; Y = 4;
  return X + Y + a + b;
};
int res = fun(1,2);
// res == 11
// X == 4
// Y == 2
```

C++14 extends the C++11 lambda functions with generic lambda functions by allowing a lambda to have generic arguments:

```
auto plus = [](auto a, auto b){ return a + b;
    };
// plus(1, 2) == 3
// plus(std::string("a"),
//      std::string("b")) == "ab"
```

Finally, a classic function can take a lambda function as argument. It allows for example to create abstractions that map a function on every element of a container. For example, in Video++ it allows to apply a kernel function on all the pixels of an image.

```
template <typename T, typename F>
void map(std::vector<T>& v, F f)
{
  for (int i = 0; i < v.size(); i++)
    f(v[i]);
}

std::vector<int> v = {1,2,3,4};
map(v, [](int& a) { a += 1; });
// v == {2,3,4,5};
```

B. Variadic Templates

In C++11, a variadic template [12] takes a variable number of parameters. Recursion is often used to iterate on the list of arguments.

```

// A function processing a list of argument

template <typename T, typename ...Tail>
inline void fun() {} // Case with 0 argument.

template <typename T, typename ...Tail>
inline void fun(T arg, Tail... tail) // Case
    with N arguments
{
   // process arg...
   // then process the tail
   fun(tail...); // Call fun on N-1 arguments.
}
```

Video++ uses variadic templates to build abstractions that take a variable number of arguments. For example, the `pixel_wise` construct (See sec. II-B) takes a variable number of images.

C. Range-Based For Loops

The C++11 standard also adds range-based for loops [13], a shorthand to loop over a whole container.

```
std::vector<int> v = {1,2,3,4};
for(auto& e: v) e += 1;
```

It applies on the container of the standard library and on any container that implements a minimal container interface. *Video++* uses this construct for example to easily iterate on image domains. Note that this notation generates a single threaded loop.

D. OpenMP4 and Cilk™Plus

With the raw C++ language, the programmer needs to manually spread threads over all cores, and to use vector intrinsics that are architecture specific (SSE, AVX for x86, and NEON for ARM).

OpenMP4 [14] and Cilk™Plus [15] are two frameworks that enable to take advantage of the growing number of cores and size of SIMD vectors available in a processor. With these frameworks, some special constructs allow to express the parallelism of a loop without binding the code to a specific vector instruction set.

Listing 3. OpenMP and Cilk™Plus

```
// OpenMP 4
#pragma omp parallel simd for
for(int i = 0; i < 100; i++)
  a[i] = b[i] + c[i];

// Cilkplus
a[0:100] = b[0:100] + c[0:100]
```

II. The *Video++* Library

This section presents the main principles of the framework *Video++*.

A. Image Types and Domains

The *Video++* `imageNd<T, N>` type holds a multi-dimensional array of dimension N containing values of type T. N and T are templates parameters. The domain of an image is an N-hyper-rectangle accessible via the `domain` method. `image2d<V>` and `image3d<V>` are simple aliases to `imageNd <V, 2>` and `imageNd<V, 3>`.

The assignment operator shares the source data with destination. It allows to avoid deep copies of images when the container is passed by value to a function. Also, thanks to the C++11 shared pointers, the library automatically frees memory when an instance of image data is not referenced anymore.

```
{
  // Allocates a 100 rows x 200 cols image of
      int values.
  image2d<int> img(100, 200);

  // Initializes image values.
  fill(img, 0);

  // Access to the value of the first pixel.
  int pixel0 = img(0,0);

  // Assignment shares the data between two
      containers.
  image2d<int> img2 = img;

  // img2 and img now share the same buffer:
  img2(0,0) = 42;
  // img(0,0) == img2(0,0)

  // Clones an image to duplicate its data.
  image2d<int> img3 = clone(img);
  // img3 holds its own separate buffer.

  // Adds an extra border of 2 pixels, useful
      to speed up filters accessing
      neighborhoods.
  image2d<int> img(100, 200, border(2));

}
// The two allocated images are automatically
     freed at the end of the scope.
```

To speed up access to the pixels, *Video++* ensures that the beginning of each row is aligned on 128 bits by padding, if needed, the end of each line.

B. Parallel Image Kernels

The main feature of *Video++* is the easy definition of parallel image processing kernels. The framework allows to define kernels that run up to 32 times faster (see sec. III) than the naive non optimized version, while being shorter to write.

The `pixel_wise` construct allows to spread the execution of a kernel over all the available CPU cores. Since it splits the execution in several threads running on several cores, there must be no dependency between the computation of two pixels.

`pixel_wise` takes a variable number of objects (images or domains) and builds a kernel runner that will run a lambda function via `operator<<`. The number of arguments of the lambda function and `pixel_wise` must be equal. All the containers must have the same domain of definition. For readability, we overloaded `operator<<` so the kernel function does not lengthen the argument list of `pixel_wise` excessively.

The kernel runner passes to the lambda function each n-tuple of elements sharing the same coordinates such that the *i*th argument passed to the function will be an element of the *i*th container. In listing 4, references `a`, `b` and `c` will respectively take references to pixels of images `A`, `B` and `C`.

Listing 4. pixel_wise kernel

```
image2d<int> A(100, 100);
image2d<int> B(A.domain());
image2d<int> C(A.domain());

fill(B, 2);
fill(C, 3);

pixel_wise(A, B, C) << // Parallel iteration
    on all pixels of A, B and C.
  [] (int& a, int& b, int& c) {
    // a, b and c are references to pixels of
        A, B and C.
    a = b + c;
  };

// A is now filled with 5.
```

Behind the scene, `pixel_wise` uses two optimizations:

- OpenMP parallel loops, to fully exploit all cores available. Each OpenMP thread processes at least one image row such that two different threads do not write on the same cache line: It would cause false sharing [16] and degrade performances.
- 1D pointer arithmetic, to iterate on image buffers. It provides a significant speedup over the traditional 2D image addressing such as `image[row][col]` or `image[row * n_cols + col]`.

C. Accessing Neighborhood

Video++ provides fast access to neighbor pixels. This section explains how to access the neighborhoods.

The `box_nbh2d` construct allows to process all (or part of) the pixels in a rectangle window centered on the current p, where the size of the rectangle is set at compile time.

Note that the size of the neighborhood is passed as template parameter such that the compiler can better optimize the assembly with loop unrolling and vectorization. Listing 5 shows the implementation of a 5x5 box filter. The benchmark of section III-B shows that this construct offers a significant speedup.

Listing 5. A 5x5 box filter

```
image2d<int> A(1000, 1000, border(2)); // The input.
image2d<int> B(A.domain(), border(2)); // The output.

auto Bnbh = box_nbh2d<int, 5, 5>(A);

// Parallel Loop over pixels of A and B.
pixel_wise(A, Bnbh) <<
[] (int& a, auto& b_nbh) {
  int sum = 0;

  // Sum the whole 5x5 neighborhood.
  b_nbh.for_all([&sum] (int& n) {sum += n;});

  // Write the sum to the output pixel.
  b = sum / 25;
};
```

D. Interoperability with OpenCV

The header `vpp/opencv_bridge.hh`, not included by default, provides conversions to and from OpenCV image type `cv::Mat`. It allows to run *Video++* code on OpenCV images and *vice versa*, without cloning the pixel buffer.

Responsibility for freeing the data will switch to OpenCV or *Video++* depending of the order of the destructor calls.

```
// Load JPG image in a vpp image using OpenCV imread.
image<vuchar3> img = from_opencv<vuchar3>
                (cv::imread("image.jpg"));

// Write a vpp image using OpenCV imwrite.
cv::imwrite("in.jpg", to_opencv(img));
```

E. Vector Types

Even though users of the library can store any pixel type inside the image containers, *Video++* provides aliases to Eigen3 vector types for users who need linear algebra operators on pixel values. The Eigen3 library provide type-safe generic vectors and matrix type and linear algebra primitives. It is highly optimized thanks to the use of SIMD vector extensions (SSE, AVX, NEON) available today on many processors.

The vector types are noted `vTN` where:

- T is one the following: **char, short, int, float, double, uchar, ushort, uint**.
- N is an integer between 0 and 4.

For exemple, `vint2`, `vuchar3`, `vfloat4` are valid vector types and the type `image2d<vuchar4>` can handle an RGBD 8-bit image.

III. Benchmarks

In programming languages, abstractions allow to write a smaller and less error-prone code. However, the resulting code often runs slower than a homologous manually optimized program. In this section, we compare implementations of two image processing loops and show that the proposed framework allows to shorten the code without impacting its performances. The benchmark also compares *Video++* with the OpenCV equivalent functions. These experiments ran on a 4 hyper-threaded 3.2GHz cores Intel i7-4700HQ with AVX2 vector extensions. All the executables were compiled with GCC 4.9.1 and the flags `-O3 -march=native`.

A. A Pixel Wise Filter: Image Addition

Many image processing filters are simple functions that fill pixels with computed values, thus featuring no dependencies between computations regarding different pixels. This section compares three versions of one of them: Adding two images, or `A(r,c)= B(r,c)+ C(r,c)`

The first one is the naive version. It iterates on rows and columns with two for loops and uses 2D addressing to access pixels:

Listing 6. image add, naive version

```
for (int r = 0; r < A.nrows(); r++)
  for (int c = 0; c < A.ncols(); c++)
  {
    A(r, c) = B(r, c) + C(r, c);
  }
```

The second one uses the `pixel_wise` construct:

Listing 7. image add, pixel wise version

```
pixel_wise(A, B, C) << [] (int& a, int& b,
                           int& c)
{
  a = b + c;
};
```

The third one is the more verbose, manually optimized loop that we want to match in terms of running time. It leverages a fast 1D addressing, multi-threading, and SIMD vector extensions:

Listing 8. image add, raw OpenMP version

```
int nr = A.nrows();
#pragma omp parallel for
for (int r = 0; r < nr; r++)
{
  int* curA = &A(r, 0);
  int* curB = &B(r, 0);
  int* curC = &C(r, 0);
  int* endA = curA + A.nrows();

  int nc = A.ncols();
#pragma omp simd
  for (int i = 0; i < nc; i++)
  {
    curA[i] = curB[i] + curC[i];
  }
}
```

Figure 1 and figure 2 show that the performances of the Video++ `pixel_wise` and the raw OpenMP versions are very close, while running 2x to 26x faster than the naive version, and slighly faster than the OpenCV routine `cv::add`.

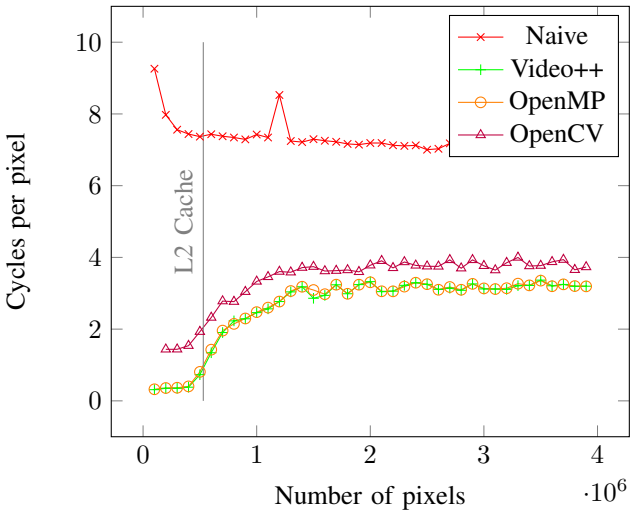

Fig. 1. Cycles per pixel of the naive, pixel wise raw OpenMP and OpenCV versions of image add with different image sizes. The three images fully fits in the 6.2MB L2 cache only at the left of the vertical line.

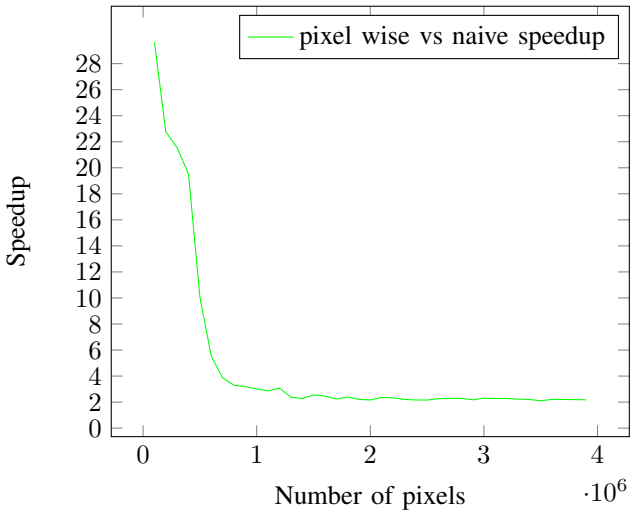

Fig. 2. Speedup of the pixel wise image add over the naive image add.

B. A Convolutional Filter: The Box Filter

Using the same protocol that in the previous section, we compare four versions of a box filter. The naive version launches four nested loops to iterate over the rows, columns and neighbors. It accesses image values via 2D addressing:

Listing 9. box filter naive version

```
for (int r = 0; r < A.nrows(); r++)
for (int c = 0; c < A.ncols(); c++)
{
  int sum = 0;
  for (int d = -2; d <= 2; d++)
  for (int e = -2; e <= 2; e++)
    sum += B(r + d, c + e);
  A(r, c) = sum / 25;
}
```

The shortest version (See listing 5) combines `box_nbh2d` and `pixel_wise`. We wrote an other version by manually optimizing the loops using OpenMP:

Listing 10. box filter OpenMP version

```
int nr = A.nrows();
#pragma omp parallel for
for (int r = 0; r < nr; r++)
{
  int* curA = &A(vint2(r, 0));
  int* curB = &B(vint2(r, 0));
  int* endA = curA + A.nrows();

  int nc = A.ncols();
  int sum = 0;

  int* rows[5];
  for (int i = -2; i <= 2; i++)
    rows[i + 2] = &B(vint2(r + i, 0));

  for (int i = 0; i < nc; i++)
  {
    int sum = 0;
    for (int d = -2; d <= 2; d++)
      for (int e = -2; e <= 2; e++)
        sum += rows[d + 2][i + e];
    curA[i] = sum / 25;
  }
}
```

As in the previous benchmark, the OpenMP and *Video++* versions have the same running time while the naive version is 16 to 32 times slower. The OpenCV `cv::boxFilter` is 5.6 times slower because it is single threaded.

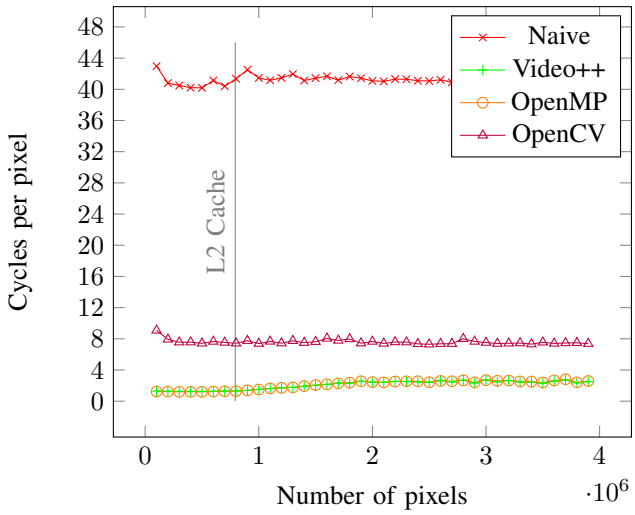

Fig. 3. Running times of the four implementations of a box filter with different number of pixels. The two images fully fits in the 6.2MB L2 cache only at the left of the vertical line.

IV. Discussion and Conclusive Remarks

Although we ran the benchmark on a 4-hyperthreaded cores, the library could scale on processors with more cores

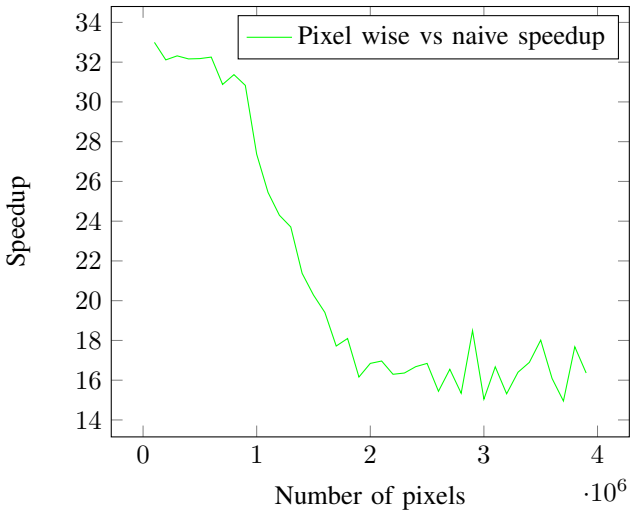

Fig. 4. Speedup of the pixel wise box filter over the naive version.

since OpenMP supports it. Furthermore, since we let the compiler unroll and vectorize the loops, there is no restriction on the vector instruction set as long as the compiler implements the necessary optimizations.

Video++ provides a set of generic objects and routines that allow to write efficient implementations of simple filters quickly. At the moment, it is actually relevant only for local and regular operations and only targets multi-cores CPUs.

As of today, the library introduced in this paper is a set of tools that ease the writing of fast image processing applications on multi-core CPUs. In a near future, we plan to extend the library to support other architectures like GPUs (using tools like C++AMP [17]) and provide fast implementations of more filters, point detectors, and feature tracking.

We proposed in this paper a new open source framework to build fast image and video processing applications. It heavily relies on the new C++ standards C++11 and C++14 that are now available in major compilers like GCC [1] or Clang [2]. We showed that while providing abstractions that allow to develop applications faster, it does not make any compromise in terms of performances. Every provided abstraction runs at exactly the same speed as the manually optimized code and up to 32 times faster than the naive version.

Acknowledgment

This work has been funded by the French DGA (General Directorate for Armament).

References

[1] The Free Software Foundation Inc. GCC, the GNU Compiler Collection. [Online]. Available: http://gcc.gnu.org/

[2] Apple Inc. Clang: a C language family frontend for LLVM. [Online]. Available: http://clang.llvm.org/

[3] G. Bradski, "The OpenCV library," *Dr. Dobb's Journal of Software Tools*, 2000.

[4] N. Burrus, A. Duret-Lutz, T. Géraud, D. Lesage, and R. Poss, "A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming," in *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*, Anaheim, CA, USA, Oct. 2003.

[5] U. Köthe, *Reusable software in computer vision*. Academic Press, 1999.

[6] D. Tschumperlé. The CImg Library. [Online]. Available: http://cimg.sourceforge.net/

[7] L. Ibáñez, W. Schroeder, L. Ng, and J. Cates, *The ITK Software Guide*. Kitware, 2003.

[8] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2013, pp. 519–530.

[9] M. Garrigues. Video++ Git Repository. [Online]. Available: https://github.com/matt-42/vpp

[10] D. Vandevoorde. New wording for c++0x lambdas (rev. 2). [Online]. Available: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2927.pdf

[11] V. Faisal, S. Herb, and A. Dave. Generic (polymorphic) lambda expressions (revision 3). [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3649.html

[12] G. Douglas, J. Jaakko, and M. Jens. Proposed wording for variadic templates (revision 2). [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2242.pdf

[13] D. Gregor. Range-based for loop wording. [Online]. Available: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2930.html

[14] OpenMP Architecture Review Board. Openmp application program interface, version 4.0. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[15] Intel Corporation. Cilk™ plus. [Online]. Available: http://www.cilkplus.org/

[16] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance," in *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems-Volume 4*. USENIX Association, 1993, pp. 3–3.

[17] Microsoft. C++ Accelerated Massive Parallelism. [Online]. Available: http://msdn.microsoft.com/en-us/library/hh265137.aspx