OpenMP is a collection of compiler directives and library functions that are used to create parallel programs for shared-memory computers.
The \MP" in OpenMP stands for \multi-processing", another term for shared-memory parallel computing.
OpenMP is combined with C, C++, or Fortran to create a multithreading programming language, in which all processes are assumed to share a single address space.
OpenMP is based on the fork / join programming model: all programs start as a single (master) thread,
fork additional threads where parallelism is desired (the
parallel region), then join back together.
This sequence of events is repeated until all the parallel
regions have been executed.
Note: The threads must synchronize before joining.

The philosophy of OpenMP is to not sacri_ce ease of coding and maintenance in the name of performance.
Accordingly, OpenMP was designed based on two principles: sequential equivalence and incremental parallelism.

A program is said to be sequentially equivalent if it returns the same results whether it executes on one thread or many threads.
Such programs are generally easier to understand, write, and hence maintain.
Incremental parallelism is the process of taking working serial code and converting pieces to execute in parallel.
At each increment, the code can be re-tested to ensure its correctness, thus enhancing the likelihood of success for the overall project.
Note that although this process sounds appealing, it is not universally applicable.

Recall the \hello, world!" program in Fortran90:

```
PROGRAM helloWorld
PRINT *, "hello, world"
END PROGRAM helloWorld
```

We have hinted that OpenMP is explicitly parallel: Any parallelism in the code has to be put there explicitly by the programmer.
The good news is that the low-level details of how the parallelism is executed is done automatically.
In Fortran, it is easy to denote that the PRINT statement should be executed on each thread by enclosing it in a block governed by a compiler directive.

```
PROGRAM HelloWorldOpenMP

!$OMP PARALLEL
PRINT*, "hello, world"
!$OMP END PARALLEL
END PROGRAM HelloWorldOpenMP
```

The program can be compiled on a shared-memory machine (like moneta.usask.ca) via
gfortran -fopenmp helloWorldOMP.f90 -o helloWorldOMP
and executed with
./helloWorldOMP

On moneta this produces the output

```
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
hello, world
```

Notes:

1. We can conclude that the default number of threads on moneta is 16.
2. It is possible to specify the number of threads (e.g.,4) by setting an environment variable via setenv OMP_NUM_THREADS 4 or from within the program via the statement
CALL OMP_SET_NUM_THREADS(4)

3. OpenMP requires that I/O be thread safe; i.e., output from one thread is handled without interference from any other threads.
However, as usual, the threads can print out in any order, depending on the order in which they reach the print command.

4. This program can be compiled and run on a serial machine (with one thread) using a serial compiler because the compiler directives are treated as comments.
5. Compiler directives using a _xed format (as per Fortran 77) can be speci_ed as

!$OMP
*$OMP
They must start in column 1; continuation lines must have a non-blank or non-zero character in column 6; comments may appear after column 6 starting with !.
Only !$OMP is available for free format. The

directive must start the line, but it may start at any column; & is the continuation marker at the end of the line; comments may appear after column 6 starting with !.

Things are slightly di_erent in C.
The compiler directives are called pragmas, with syntax
pragma
where the # appears in column 1 and the remainder of the directive is aligned with the rest of the code.
pragmas are only allowed to be one line long; so if one happens to require more than one line,

the line can be continued using n at the end of intermediate lines.

Open Computing Language (OpenCL) is an open standard for writing code that runs across heterogeneous platforms including CPUs, GPUs, DSPs and etc. In particular OpenCL provides applications with an access to GPUs for non-graphical computing (GPGPU) that in some cases results in significant speed-up. In Computer Vision many algorithms can run on a GPU much more effectively than on a CPU: e.g. image processing, matrix arithmetic, computational photography, object detection etc.

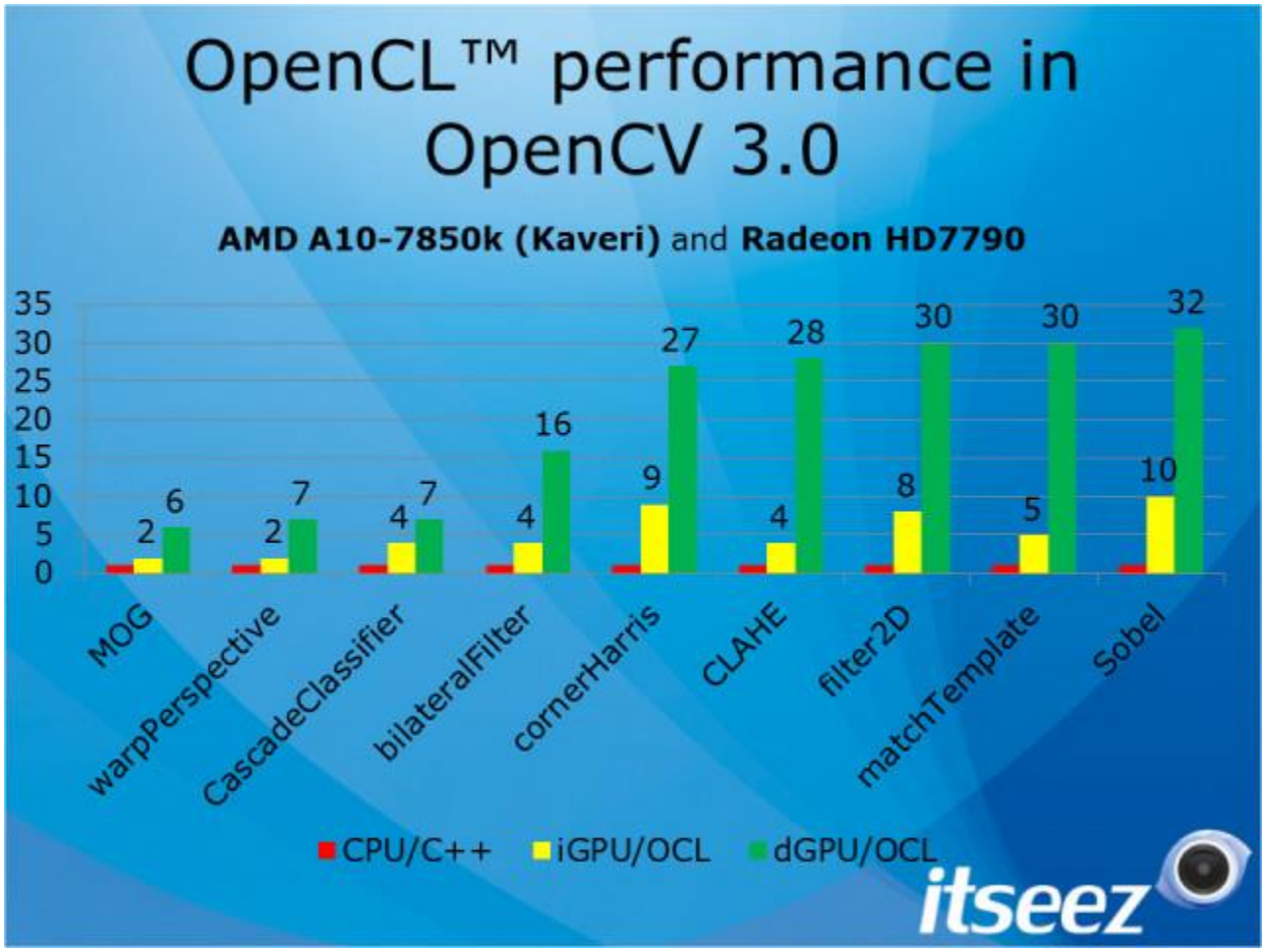
History

Acceleration of OpenCV with OpenCL started 2011 by AMD. As the result the OpenCV-2.4.3 release included the new `ocl`module containing OpenCL implementations of some existing OpenCV algorithms. That is, when OpenCL runtime and a compatible device are available on a client machine, user may call `cv::ocl::resize()` instead of `cv::resize()` to use the accelerated code. During 3 years more and more functions and classes have been added to the `ocl` module; but it has been a separate API alongside with the primary CPU-oriented API in OpenCV-2.x.

In OpenCV-3.x the architecture concept has been changed to the so-called Transparent API (T-API). In the new architecture a separate OpenCL-accelerated `cv::ocl::resize()` is removed from external API and becomes a branch in a regular `cv::resize()`. This branch is called automatically when it's possible and makes sense from the performance point of view. The T-API implementation was sponsored by AMD and Intel companies.

Numbers

Some performance numbers are shown on the picture below:



Code sample

Regular CPU code

```
// initialization
VideoCapture vcap(...);
CascadeClassifier fd("haar_ff.xml");
Mat frame, frameGray;
vector<rect> faces;
for(;;){
  // processing loop
  vcap >> frame;
  cvtColor(frame, frameGray, BGR2GRAY);
  equalizeHist(frameGray, frameGray);
  fd.detectMultiScale(frameGray, faces, ...);
  // draw rectangles …
```

```
  // show image …
}
```

OpenCL-aware code OpenCV-2.x

```
// initialization

VideoCapture vcap(...);

ocl::OclCascadeClassifier fd("haar_ff.xml");

ocl::oclMat frame, frameGray;

Mat frameCpu;

vector<rect> faces;

for(;;){

  // processing loop

  vcap >> frameCpu;

  frame = frameCpu;

  ocl::cvtColor(frame, frameGray, BGR2GRAY);

  ocl::equalizeHist(frameGray, frameGray);

  fd.detectMultiScale(frameGray, faces, ...);

  // draw rectangles …

  // show image …
}
```

OpenCL-aware code OpenCV-3.x

```
// initialization

VideoCapture vcap(...);

CascadeClassifier fd("haar_ff.xml");

UMat frame, frameGray;

vector<rect> faces;

for(;;){

  // processing loop

  vcap >> frame;

  cvtColor(frame, frameGray, BGR2GRAY);

  equalizeHist(frameGray, frameGray);

  fd.detectMultiScale(frameGray, faces, ...);
```

```
  // draw rectangles …

  // show image …
}
```

Cilk development at MIT

The Cilk programming language grew out of three separate projects at the MIT Laboratory for Computer Science:

- Theoretical work on scheduling multi-threaded applications
- StarTech – a parallel chess program built to run on the Thinking Machines Corporation's Connection Machine model CM-5
- PCM/Threaded-C – a C-based package for scheduling continuation-passing-style threads on the CM-5

In April 1994 the three projects were combined and christened Cilk. The name "Cilk" is not an acronym, but an allusion to "nice threads" (silk) and the C programming language.

The Cilk-1 system was released in September 1994. The current implementation, Cilk-5.3, is an extension of ANSI C and is implemented as a source-to-source translator. Cilk-5.3 is available from the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), though it is no longer supported. Cilk-5 allocates the frame of a Cilk function on the heap, requiring the use of the `spawn` keyword to call a Cilk function, and the `cilk` keyword on Cilk function declarations. The MIT releases are sometimes referred to as "MIT Cilk."

Cilk++

In 2006, Cilk Arts licensed the Cilk technology from MIT with the goal of developing a commercial C++ implementation. Cilk++ v1.0 was released in December 2008 with support for both Windows* Visual Studio and GCC/C++ on Linux. Cilk++ differed from Cilk-5 in the following ways:

- Full C++ support, including exceptions
- C++ code can call Cilk code directly, as long as it is compiled with the Cilk++ compiler and has Cilk linkage
- Renamed `spawn` and `sync` keywords to `cilk_spawn` and `cilk_sync` to avoid naming conflicts
- Added `cilk_for` loops to parallelize loops over a fixed number of entries
- Added "reducer hyperobjects" to help programmers deal with races caused by parallel accesses to global variables in a lock-free manner

Like Cilk-5, Cilk++ allocates Cilk function frames from the heap. While a Cilk function can call or spawn Cilk, C or C++ functions, C or C++ functions compiled with a standard compiler cannot directly call a Cilk function.

The Cilk++ kit includes the Cilkscreen race-detection tool as well as the Cilkview scalability analyzer.

Intel Cilk Plus

In 2009, Intel Corporation acquired Cilk Arts. The Cilk technology was merged with Array Notation to provide a comprehensive language extension to implement both task and vector parallelism. Intel Cilk Plus was released by Intel in 2010 as part of the Intel C++ Composer XE compiler. Key features include:

- Supports both C and C++
- Compatibile with standard debuggers
- Uses standard calling conventions - Cilk function frames are allocated on the stack so C/C++ functions can call Cilk functions freely

Intel has made the Intel Cilk Plus specifications freely available on the web.

In 2011, Intel announced that it was implementing Intel Cilk Plus in the "cilkplus" branch of GCC. The initial implementation was completed in 2012 and presented at the 2012 GCC Tools Cauldron conference. Intel has also proposed Intel Cilk Plus as a standard to the C++ standard body.

Cilk is a C/C++ extensions to support nested data and task parallelisms

• The Programmers identify elements that can safely be executed in parallel cilk threads\ data parallelism\– Nested loops cilk threads\ task parallelism\– Divide-and-conquer algorithms

• The run-time environment decides how to actually divide the work between processors – can run without rewriting on any number of processors

Important Features of Cilk

• Extends C/C++ languages with six new keywords – cilk, spawn & sync – inlet & abort – SYNCHED

• Has a serial semantics

• Provides performance guarantees based on performance abstractions.

• Automatically manages low-level aspects of parallel execution by Cilk's runtime system. – Speculation – Workload balancing (work stealing)

Basic Cilk Programming (1)

• C/C++ extensions to support nested task and data parallelism

• Fibonacci example Sequential version

```
int fib (int n) {
if (n < 2)
return 1;
 else
{
int rst = 0;
rst += fib (n-1);
rst += fib (n-2);
 return rst;
 }
 }
```

Basic Cilk Programming (2)

• C/C++ extensions to support nested task and data parallelism

• Fibonacci example

```
 Pthread version
arg_structure;
void * fib(void * arg);
 pthread_t tid;
 pthread_create(tid, fib, arg); …
 phread_join(tid);
 pthread_exit;
}
```

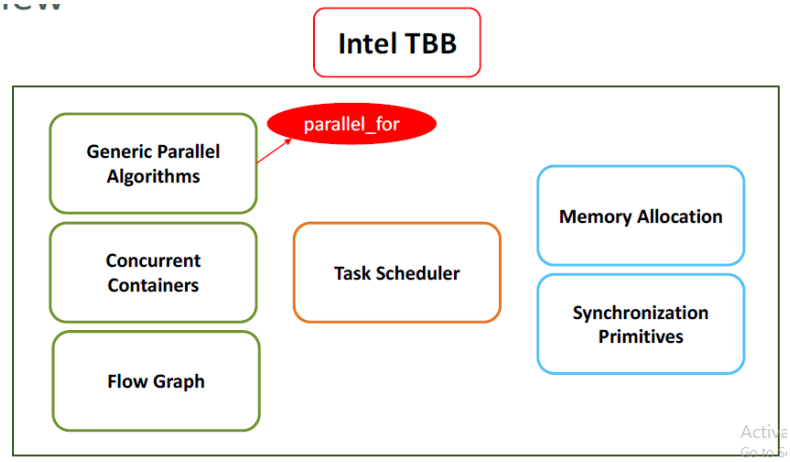
Sequential version

```
 int fib (int n)
{
 if (n < 2)
 return 1;
 else
 {
```

```
 int rst = 0;
 rst += fib (n-1);
rst += fib (n-2);
return rst;
 }
}
```

What is TBB?

- TBB is a library that supports scalable parallel programming using standard C++ code.
  - Specify logical parallelism instead of threads
  - Target threading for robust performance
  - Emphasize on scalable, data-parallel programming
  - Shared memory
  - Portable and open source



Code Example

Sequential code:

```
void SerialApplyFoo( float a[ ], size_t  n) {
      for( size_t  i=0;  i!=n;  ++i)
            Foo(a[i]);
}
```

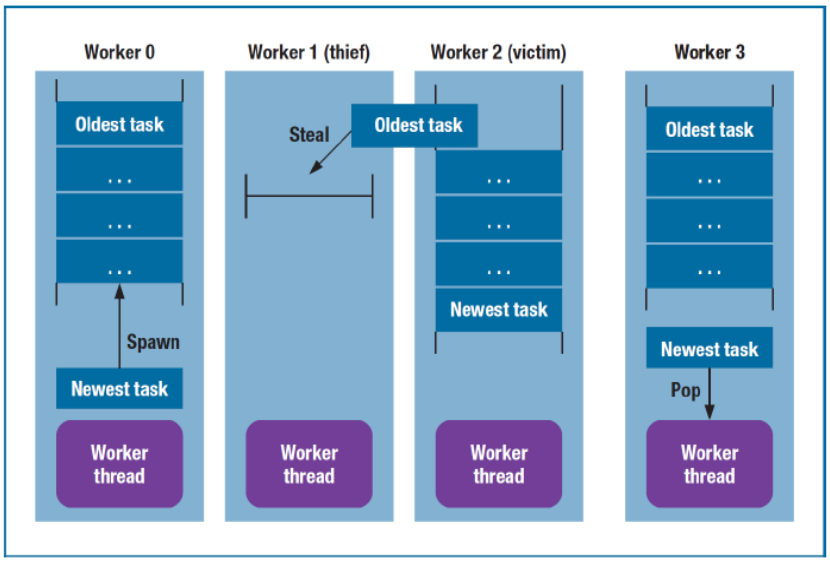
Parallel code:

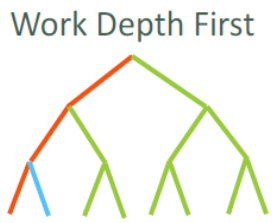
```
#include "tbb/tbb.h"
using namespace tbb;
class ApplyFoo {
      float *const my_a;
public:
      void operator() ( const blocked_range<size_t>& range ) const {
            float *a = my_a;
            for( size_t i = range.begin(); i!= range.end(); ++i )
                  Foo(a[i]);
      }
      ApplyFoo( float a[] ) :  my_a(a) {}
};

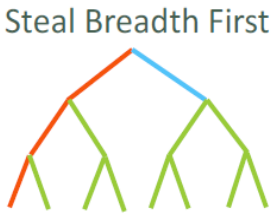
void ParallelApplyFoo( float a[], size_t n ) {
      parallel_for ( blocked_range<size_t>(0,n),  ApplyFoo(a) );
}
```

Body object

Dynamic Task Scheduler – *Task Stealing*

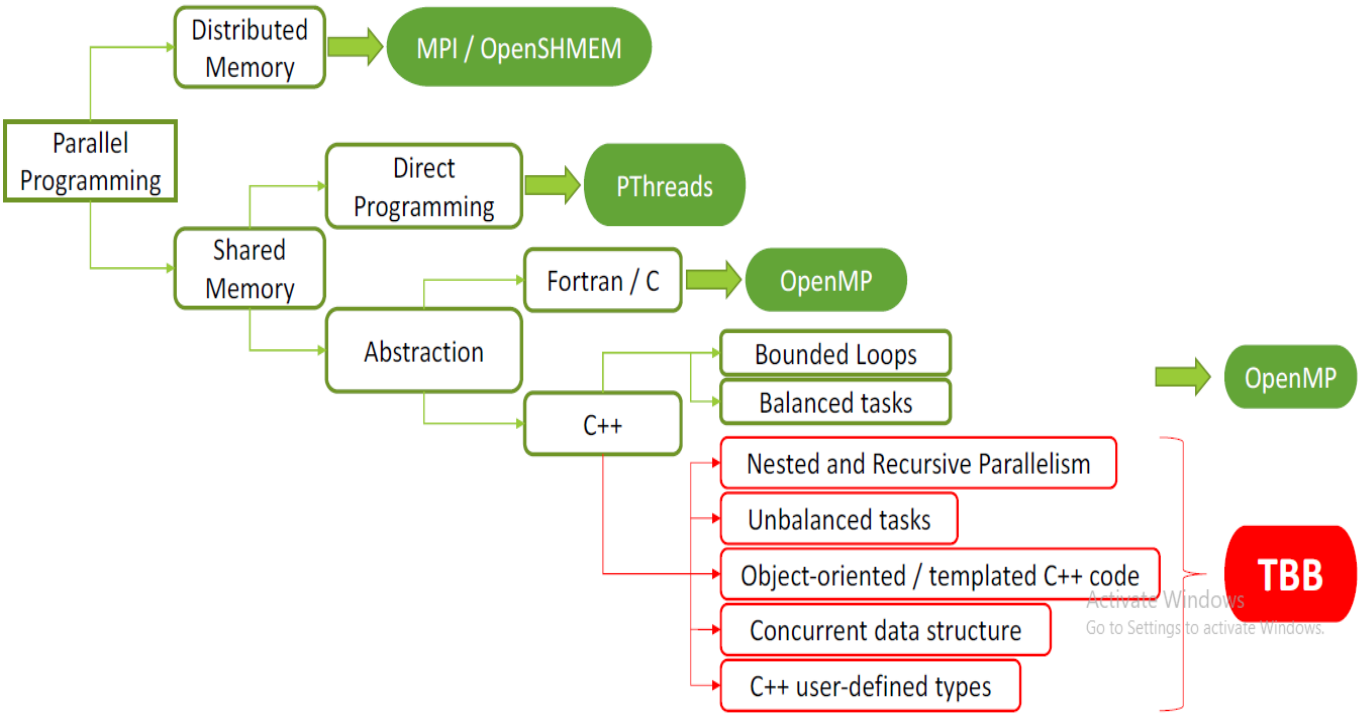






Activate Windo
Go to Settings to a

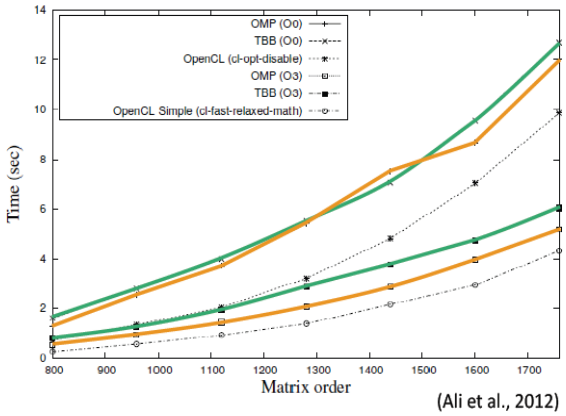
Is TBB right for you?



Activate Windows
Go to Settings to activate Windows.

TBB vs. OpenMP Performance

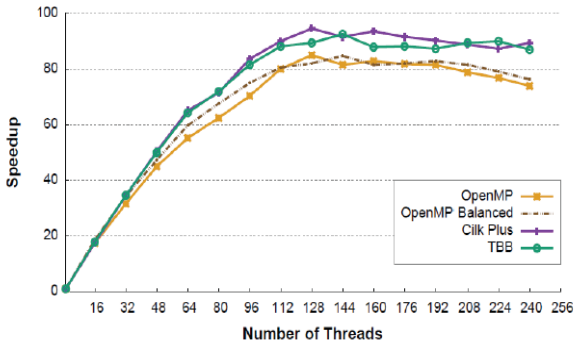
Matrix Multiplication (8 cores)



(Ali et al., 2012)

Fixed number of loop iterations
Equally distributed tasks
➡ **OpenMP**

Fibonacci 47 (cutoff 2048)



(Tousimojarad and Vanderbauwhede, 2014)

Nested and Recursive Parallelism
Unbalanced tasks
➡ **TBB**

Activate Windows
Go to Settings to activate Windows.

Unlike previous generations that partitioned computing resources into vertex and pixel shaders, the CUDA Architecture included a unified shader pipeline, allowing each and every arithmetic logic unit (ALU) on the chip to be marshaled by a program intending to perform general-purpose computations. Because NVIDIA intended this new family of graphics processors to be used for generalpurpose computing, these ALUs were built to comply with IEEE requirements for single-precision floating-point arithmetic and were designed to use an instruction set tailored for general computation rather than specifically for graphics. Furthermore, the execution units on the GPU were allowed arbitrary read and write access to memory as well as access to a software-managed cache known as shared memory. All of these features of the CUDA Architecture were added in order to create a GPU that would excel at computation in addition to performing well at traditional graphics tasks.

. The following represent just a few of the ways in which people have put CUDA C and the CUDA Architecture into successful use.

1.5.1 MEDICAL IMAGING

The number of people who have been affected by the tragedy of breast cancer has dramatically risen over the course of the past 20 years. The mammogram, one of the current best techniques for the early detection of breast cancer, has several significant limitations. Two or more images need to be taken, and the film needs to be developed and read by a skilled doctor to identify potential tumors. Additionally, this X-ray procedure carries with it all the risks of repeatedly radiating a patient's chest. After careful study, doctors often require further, more specific imaging—and even biopsy—in an attempt to eliminate the possibility of cancer. These false positives incur expensive follow-up work and cause undue stress to the patient until final conclusions can be drawn.

1.5.2 COMPUTATIONAL FLUID DYNAMICS

For many years, the design of highly efficient rotors and blades remained a black art of sorts. The astonishingly complex movement of air and fluids around these devices cannot be effectively modeled by simple formulations, so accurate simulations prove far too computationally expensive to be realistic. The availability of copious amounts of low-cost GPU computation empowered the Cambridge researchers to perform rapid experimentation. Receiving experimental results within seconds streamlined the feedback process on which researchers rely in order to arrive at breakthroughs. As a result, the use of GPU clusters has fundamentally transformed the way they approach their research. Nearly interactive simulation has unleashed new opportunities for innovation and creativity in a previously stifled field of research.

1.5.3 ENVIRONMENTAL SCIENCE

The increasing need for environmentally sound consumer goods has arisen as a natural consequence of the rapidly escalating industrialization of the global economy. Growing concerns over climate change, the spiraling prices of fuel, and the growing level of pollutants in our air and water have brought into sharp relief the collateral damage of such successful advances in industrial output. Detergents and cleaning agents have long been some of the most necessary yet potentially calamitous consumer

products in regular use. As a result, many scientists have begun exploring methods for reducing the environmental impact of such detergents without reducing their efficacy.