Fall 12-1-2016

A Build System for Benchmarking and Comparison of Competing System Implementations

Anthony J. Bentley
University of New Mexico

Anthony J. Bentley

Candidate

Electrical & Computer Engineering

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

L. Howard Pollard , Chairperson

Daryl Lee

James Plusquellic

A Build System for Benchmarking and Comparison of Competing System Implementations

by

Anthony J. Bentley

B.S., Electrical Engineering, University of New Mexico, 2012

THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science
Computer Engineering

The University of New Mexico

Albuquerque, New Mexico

December, 2016

Acknowledgments

I am indebted to many members of the Advanced RF group at Sandia National Laboratories for their assistance and for providing both software licenses and computing hardware for this thesis, as well as the algorithm described in Chapter 4.

I would like to thank my advisor, Dr. L. Howard Pollard, for his advice, his wealth of experience, and his patience.

A Build System for Benchmarking and Comparison of Competing System Implementations

by

Anthony J. Bentley

B.S., Electrical Engineering, University of New Mexico, 2012

M.S., Computer Engineering, University of New Mexico, 2016

Abstract

When developing a hardware or software system, the problem at hand may lend itself to multiple solutions. During the implementation process for such systems, it can be helpful to prototype multiple versions that use distinct paradigms, and determine the efficiency of each according to some metric, such as execution time. This paper presents a portable, lightweight build system designed for easy benchmarking and verification of competing implementations of an algorithm. Also presented is a sample project that uses this system to compare the performance and correctness of CPU, GPU, and FPGA implementations of a signal recovery algorithm.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivations

Much time is spent, and reams of paper written, on the topic of software development processes. Often a lot of effort is spent discussing management processes and "people"-involved development methodologies. But experience shows that design of the *code* can also be a significant factor impacting the usability of software—in particular, by developers.

With few exceptions, software must be built before it can be executed. Build systems that turn a collection of source code to executable programs vary dramatically in complexity from project to project, from a simple list of commands to preprocessors that automatically scan code to develop extensive dependency trees and distribute compilation work between multiple CPUs. The build system used by a project has an impact on how code is built—but it can also have an impact on the act of developing the code.

When a project is being built from infancy, a build system that enforces a modular design from the ground up can provide a useful platform to determine questions such as, "What hardware targets should I use?" "Is this algorithm faster than that one?" Is letting the build system enforce design worth it? This thesis demonstrates so by presenting such a build system and implementing a real-world design on top of it.

Chapter 2 presents a set of design constraints for a build system, especially portability and lightweight footprint. It defines portability in terms of compliance with POSIX, the IEEE standard for operating system interfaces. It focuses on the C library and POSIX-defined `make` interface, continuing with a survey of build systems and tools commonly used in software development. The chapter concludes with a summary of lessons learned from using these build systems and settles on a more complete set of design constraints.

Chapter 3 introduces a `make`-based build system designed according to the principles developed in Chapter 2, in order to facilitate quick prototyping and benchmarking of new programs that have been designed in a modular fashion. It also describes the facilities that POSIX provides for accurate timekeeping, and accommodations that must be made in a software project to take advantage of them.

Chapter 4 describes a real-world signal recovery program, purposely written to take advantage of the build and benchmarking features described in previous chapters. The specifics of developing multiple parallel implementations—CPU, GPU, and FPGA—are also discussed. The chapter ends with a performance comparison between the CPU and GPU implementations of the algorithm.

Chapter 5 concludes with lessons learned during the design and implementation of the build system, and the additional knowledge gleaned during the process of designing software to use it.

Chapter 2

Build system design

2.1 Foundation: Initial design requirements

The fundamental purpose of a build system is to *build* a project—to generate a final product from a collection of (typically human-editable) source files. The source files may be anything; typically, they are a sequence of commands in some programming language that in the end are converted into machine-executable computer code, but a build system may also convert graphics or audio content from one format to another, or perform text processing on documentation, or any number of possibilities.

A number of factors go into the design of any software project. Its interface must be adequate for the target audience to be able to use it effectively. Obviously, the software must *work* (i.e., correctly perform the task given). And it must be designed in such a way as to properly execute in the target environments that the software will run in, such as common desktop operating systems or a user's web browser.

Similarly, the build system's primary design requirements come from the fact that it must be configurable by a human (*interface*), and that it must work properly (*correctness*), across a variety of target environments (*portability*).

2.1.1 Interface

Typical examples of configurability in a software project include enabling or disabling functionality, enabling alternate implementations of internal code, or customizing the build based on characteristics of the target system.

But too much configurability can be a *bad* thing! If users are encouraged (or required) to manually configure the build, human error is likely to introduce bugs that result in wasted time spent debugging.

Thus, an important factor of a build system's user interface is how much of the build process is automated. Why automate? What are the benefits? They are twofold: less chance for human error, and a simple universal interface means new developers can get up to speed quickly without training.[13]

2.1.2 Correctness

Reproducibility is a major component of correctness.

A build should generate complete output from a minimal number of inputs. If an output file o depends on a generated file g, which is generated from an input file i, the build system should be able to reproduce o from i even if g has been deleted.

Having complete start-to-finish builds reduces the burden of storing intermediate files in version control, which often are large and typically contain little human-readable output.

Another factor of correctness is the importance of builds being free of errors. Build systems often make use of parallelism and concurrency by utilizing multiple CPU cores to compile distinct blocks of code at the same time. In such an environment, a bug that triggers a race condition may cause serious problems. A build system should make use of data structures that prevent these race conditions from happening.[16]

2.1.3 Portability

Although portability is a common concern, it is also frequently overlooked and underimplemented. For a given project, builds may occur on operating systems that provide similar interfaces to each other but different capabilities and extensions. It's impractical to comprehensively test software in all such environments. However, carefully writing code that avoids the use of extensions and instead relies solely on well-defined behavior provides *confidence* that the software will work correctly on standards-compliant platforms—which is, after all, the whole point of standards.

On the other hand, what software development strategies could *not* be considered reasonably portable? One method that is widely used, but often leads to unnecessarily platform-dependent code, is the `#ifdef` directive of the C preprocessor.

In a C program, 'groups' (sections) of code may be preceded by `#ifdef` specifying some condition, and followed with a matching `#endif`. If the condition evaluates to true, the group is compiled as normal. Otherwise, the group is ignored.[11] C programs often wrap explicitly non-portable segments of code in `#ifdef` with a condition that is only true on a platform capable of understanding that code.

The authors of Plan 9 from Bell Labs, a network-based operating system and spiritual successor to Unix, criticized conditional compilation with `#ifdef` for making software difficult to read and for creating multiple divergent codepaths that increase required testing surface. If a program contains many `#ifdef`s with a variety of conditions, possibly adjacent or nested within each other, most programmers will have trouble knowing at a glance which conditions apply to a given piece of code.

The Plan 9 authors instead advocated minimizing dependencies and isolating the few platform-dependent components of the source code to individual files, rather than scattering them throughout the codebase.[18]

A similar approach to handling non-standards-compliant environments has been advocated by others. It is sometimes known as OpenSSH-style portability (named after one of the higher-profile projects that use it). In this approach, the software provides a standards-compliant copy of an API and uses it if the system is not capable of providing a standards-compliant interface itself.[15]

What standards are important, then? The ISO C language[11] and POSIX, the Portable Operating System Interface[10] are among the most influential and widespread software standards.

The C programming language was created by Dennis Ritchie of Bell Labs during the development of the Unix operating system in the early 1970s. It is defined by a modern and generally comprehensive standard, and includes a standard library with a variety of functionality including mathematical functions.

Since the ISO took control of the standardization of C in 1989, there have been two major revisions of the language, one in 1999 and another in 2011 that superseded it. The 2011 revision remains the canonical C standard.

POSIX is a superset of ISO C: it encompasses the full C standard library plus added extensions, and also standardizes a host of operating systems–relevant behaviors that are out of scope for the ISO C standard.

The POSIX standard is maintained and actively developed by the Austin Group, a collaborative effort between the IEEE, the Open Group, and the ISO.

The portability benefits of the C and POSIX standards are numerous. The primary advantage comes from their ubiquity: C is well understood by a vast number of software engineers, and its level of hardware abstraction is relatively easy to grasp.

Meanwhile, POSIX-based systems are extremely common. In practice, POSIX is a standard most commonly followed by Unix-like operating systems, such as Linux, OpenBSD, FreeBSD, Mac OS X, and Solaris, but it also governs a number of operating systems not traditionally thought of as Unix variants, such as certain restricted versions of Windows.

Relying on a standard as a baseline for software portability is helpful. Standards committees tend to move more slowly than software developers, so a project that relies on the output of the C and POSIX committees is less likely to have large fundamental changes happen without warning; indeed, the C committee explicitly adds new features in such a way that software complying to the older versions of C never changes behavior, as long as it was written to be standards-compliant in the first place.

The C standard library, the C language itself, and the software tools specified by POSIX have many implementations; by surveying existing versions of POSIX software one can limit a build system to a widespread subset of functionality to increase portability.

POSIX provides `make`, a useful language for designing a build system. It meets our requirements of portability (it is defined by a standard and has multiple standards-compliant implementations).

2.2 `make` background

`make` has been a part of Unix since Programmer's Workbench 1.0, a 1977 release of Unix version 6 designed for software development. `make` became a prolific build tool due to Unix's success, and it continues to be a standard part of most Unix-like operating systems.

`make` works by constructing a directed acyclic graph (DAG) of build dependencies and evaluating a set of rules, specified by the programmer in a 'makefile'. Constructing such a graph manually is a lot of work; as a result, software projects often used multiple simple makefiles with subtly (or not so subtly) broken build graphs, and put up with `make`'s inadequate results (rebuilding targets unnecessarily, or removing already built results and then redundantly running `make` again). These practices were discouraged in Peter Miller's 1999 paper, "Recursive `make` Considered Harmful."[16]

Other 'meta-build systems,' such as CMake and Autotools, generate makefiles using a specification written in their own custom format. These automatically generated makefiles contain a large amount of boilerplate, but when done right using them avoids problems with poorly made DAGs.

There are many versions of `make`, both derivatives of the AT&T UNIX version and independent reimplementations exist. POSIX attempted to standardize various features of `make`, with some success. The build system described in this paper was written for POSIX compliance and tested on two `make` implementations that aim for such compliance.

2.2.1 OpenBSD `make`

OpenBSD is a Unix-like operating system descended from the Berkeley Software Distribution of Unix. The OpenBSD project was founded in 1996 by Theo de Raadt, a former developer of NetBSD, a similar Berkeley derivative. Unlike a typical Linux distribution, where system software is modular to such an extreme that even the kernel and base POSIX utilities are considered external packages, OpenBSD and other BSD derivatives primarily develop a cohesive base system that includes a kernel, hardware drivers, and system administration utilities, generally including most or all utilities required by POSIX.

The versions of `make` used by all modern BSD operating systems—namely OpenBSD, FreeBSD, and NetBSD—are all descended from a common source but have diverged over the years. Though each system simply calls its implementation `make`, they are derived from `pmake` ("Parallel `make`"), originally part of the Sprite distributed network operating system from Berkeley.[5]

Relative to other versions of `make`, OpenBSD's version has been modified a number of times to improve parallelism (such as detecting recursion to avoid exponential creation of subprocesses) and correctness (both compliance to POSIX and avoidance of certain race conditions). Over time it has also accumulated new features and extensions, including some inspired by other versions of `make`.[7]

2.2.2 GNU `make`

Development of GNU `make` began in 1988 by Richard Stallman and Roland McGrath. It has since become perhaps the most prolific `make` implementation, being the default `make` program on nearly every Linux distribution, as well as Mac OS X, and available in the package repositories of every major BSD operating system.

GNU `make` is widely used. For illustration, on OpenBSD, an operating system where GNU `make` is available as an option but is not the default version of `make`, nearly 20% of the software packages available in its repositories make use of non-portable GNU `make` features.[1]

[1] This number was retrieved from an OpenBSD 5.8 system, running the following `sqlite` queries against the local `sqlports` database:

```
$ echo 'select fullpkgpath from ports where use_gmake = 1;' |
sqlite3 /usr/local/share/sqlports | wc -l
1826
$ echo 'select fullpkgpath from ports;' |
sqlite3 /usr/local/share/sqlports | wc -l
```

2.3 Existing build systems and their problems

Before building a new project with `make`, it's instructive to look at existing build systems. What considerations were made in their design? What misfeatures do they have that should be avoided? What can be learned from projects that use these build systems? Answers to these questions can be found in the following subsections.

Excessive dependencies

Dependencies are programs or libraries that provide capabilities, functions, etc. that are then used by another program. Although technically any program that uses a simple function such as `printf()` has the standard C library as a dependency, the term does not commonly refer to what provides such fundamental features. Instead, 'dependencies' usually provide nonstandard functionality, such as:

- Text translation and internationalization (e.g., Gettext[2])
- Complicated or performance-critical mathematics (e.g., the GNU Scientific Library[3] or BLAS[4])
- File formats (e.g., Libzip[5] to support compressed .zip files)
- Toolkits for graphical user interfaces (e.g., the X Window System[6])
- Shell utilities (e.g., `xsltproc`[7] for transforming XML documents)

`9291`

The majority of the remainder are built on OpenBSD with its own `make`.

[2] `https://www.gnu.org/software/gettext/`

[3] `https://www.gnu.org/software/gsl/`

[4] `http://www.netlib.org/blas/`

[5] `http://www.nih.at/libzip/`

[6] `http://www.x.org/`

[7] `http://xmlsoft.org/XSLT/xsltproc.html`

Dependencies exist because they're practical. Existing libraries may be highly optimized, widely portable, or featureful to an extent that duplicating the functionality would require significant investment.

On the other hand, dependencies can be a problem. In some situations installing new programs can be difficult: when disks are small, when system files are stored on a read-only disk, when disks are immutable in other ways, and so on. Because dependencies themselves can have dependencies, adding even a single dependency may drastically increase the installation requirements of a system. Thus, it can be beneficial to avoid gratuitous use of dependencies.

SCons[8] and Waf[9] are examples of build systems with significantly large requirements. They rely on Python, a large programming language interpreter and runtime. Similarly, Apache Ant[10] is an example of a build system that requires the Java runtime.

Another interesting example is CMake, which is written in C++. Though the C++ standard library is not often thought of as an external dependency (unlike, for example, Python or Java), it comes with its own caveats. C++ and C are two separate languages, standardized and developed by parallel committees; although there is some collaboration between the two groups, the languages themselves are separate and C++ is without question the heavier of the two. Typical open source POSIX-compliant operating systems, such as Linux or OpenBSD, consist of a majority of C code with a few (major or minor) components written in C++.

CMake is an example of a build system that must be bootstrapped with C++. This leads to an interesting problem when a system's default C++ compiler is built from a CMake-based build system, as FreeBSD's does. Out of necessity, FreeBSD,

[8] `http://scons.org/`
[9] `https://waf.io/`
[10] `https://ant.apache.org/`

whose default C++ compiler and C++ standard library come from the LLVM project,[11] maintains its own build system for the C++ compiler composed of plain Makefiles.[12]

The build system developed in this thesis is intentionally written in a manner to avoid large dependencies, instead only using programs that are both lightweight and required by POSIX—programs such as `sh`, `sed`, and `make`. `sh` is the default command interpreter on any POSIX system, and *must* be installed for most systems to function, so depending on it is not a burden. `sed` and `make` are both "building block" programs; they are so widely used that they are usually either installed by default or are easily compiled with a simple C compiler.

By intentionally making use of only portable, standardized behavior, the build system avoids depending on any particular version or derivative of these programs; therefore, if a target operating system provides a lightweight but standards-compliant `sed` (such as the one provided by Busybox[13]), users do not have the burden of installing a heaver `sed` such as GNU's in order to build their software.

Long 'configure' stages

Despite the power and expressiveness of the `make` language, some software projects require additional features and processing in their build process that `make` does not provide. Rather than calculating configuration options and detecting system features at build time, they require the user to run a 'configure' step before initiating the build. The configuration itself is typically performed by a script that executes a series of

[11] `http://llvm.org/`

[12] The build infrastructure can be found in the FreeBSD Subversion repository under `https://svn.FreeBSD.org/base/head/usr.bin/clang/`.

[13] Busybox (`https://busybox.net`) is a lightweight collection of UNIX utilities, including `sed`, that is often used to provide the basic POSIX system utilities on performance-critical systems such as routers.

shell commands to 'feel out' the characteristics of the system (such as filesystem layout), then generates the actual instructions to be passed to `make` at build time. Two build systems that require a configure stage are GNU Autoconf[14] and CMake[15]. Autoconf in particular has seen very wide adoption; over 27% of packages in the OpenBSD repositories rely on it.[16] Some software projects develop their own custom configure script from scratch, though this is less common.[17]

To illustrate typical configure stage behavior, consider the configure script from GNU Gettext, a widely used internationalization library.[18] The source code contains a configure script (named `configure`) generated by GNU Autoconf. When executed, `configure` performs a series of checks:

```
checking for a BSD-compatible install... /usr/ports/pobj/
   gettext-tools-0.19.7/bin/install.c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... mkdir -p
checking for gawk... (cached) awk
checking whether make sets $(MAKE)... (cached) yes
```

This continues for several minutes; Gettext executes 1273 of these checks.

To better understand what these checks consist of, one can examine the "thread-safe `mkdir -p`" check above. `mkdir` is a POSIX-specified utility that creates a directory in a filesystem. By default, it can only create a directory if the parent directory already exists; for example, "`mkdir x/y/`" will only create the directory

[14] `https://www.gnu.org/software/autoconf/autoconf.html`
[15] `https://www.cmake.org/`
[16] `select fullpkgpath from ports where configure_style like 'gnu';`
[17] Approximately 2.5% of programs in OpenBSD's package repository: `select fullpkgpath from ports where configure_style like 'simple';`
[18] This analysis is based on gettext-0.19.7, available from `https://www.gnu.org/software/gettext/`.

`x/y/` if `x/` already exists; if `x/` does not exist, `mkdir` will fail with an error. The `-p` option, also specified by POSIX, tells `mkdir` to create intermediate directories. So `mkdir -p x/y/` will succeed whether `x/` exists or not; if `x/` doesn't exist, it will be created.

The check itself does the following:

1. Test if the user has specified a program to create directories; if so, exit.
2. Walk the user's executable path: for each directory in `/bin/`, `/usr/bin/`, etc., look for a program named `mkdir` or `gmkdir` (GNU mkdir).
3. For each program found, execute it with the nonstandard `--version` option, and check if the output identifies it as a GNU version of `mkdir`.
4. If a GNU `mkdir` is found, use it as the program to create directories.
5. Otherwise, use an autoconf-generated shell script to create directories.

At the end of this process, the chosen program is substituted into Autoconf-generated makefiles anywhere directories must be created.

The end result of this process is:

- If the user *manually* specifies a `mkdir`, use it.
- Otherwise, if GNU `mkdir` is found on the system, use it.
- Otherwise, use an Autoconf-generated shell script.

But why doesn't Gettext just use the system `mkdir` by default? What's so special about the GNU version?

It turns out that Gettext didn't add this check; Gettext's configure script was generated by Autoconf, which adds it to *every* build that needs to create directories. The reason is detailed in the Autoconf source code:[19] the versions of `mkdir` provided by Solaris 8 and some versions of the NeXTStep operating system have a race condition. If `mkdir -p x/y/` and `mkdir -p x/z/` are executed at the same time, and `x/` does not already exist, it is possible for both processes to detect the missing `x/` and subsequently try to create it, but only one can succeed at creating `x/`; the second one to try will fail, and will stop execution of the entire program at that point, even though creating the subdirectory `x/y` or `x/z` would succeed.

This is, of course, a bug, and one that violates the standard. POSIX clearly specifies that both programs must succeed in the above situation.[10] Any POSIX-compliant system can be expected to have a working `mkdir -p`. Newer versions of Solaris fix the race condition, and NeXTStep is no longer in common use, having last seen a release in 2000.

So why does Autoconf still check for it? Because rather than assuming a standard interface, Autoconf has a strong bias in favor of old operating systems. If Autoconf instead assumed that the operating system is POSIX-compliant, and allowed the user to specify otherwise, the configure check would instead be:

1. Test if the user has specified a program to create directories; if so, exit.
2. Otherwise, use the `mkdir` provided in the user's executable path.

This avoids several filesystem accesses and allows the build system to trust the underlying operating system. Note that this method is the same as the "OpenSSH-style portability" mentioned in Section 2.1, except with POSIX as the default target rather than OpenBSD.[15]

[19] As of Autoconf version 2.69, in the file `lib/autoconf/programs.m4`.

Moving to the "assume POSIX, and provide reimplementations otherwise" model only makes sense given a few assumptions: that POSIX compliance in commonly used operating system environments generally improves over time, and that the number of workarounds for POSIX-incompatible systems is manageable.

The benefit is simplicity and improved build performance. As mentioned above, a typical Autoconf program performs thousands of configure checks, and thousands of projects use Autoconf.

Programs that use autotools either have an extra dependency that complicates the build process (autoconf), or they must keep around a huge generated file. Common practice is to avoid keeping generated files in a source control repository. The reason is that the files are often large, and when they are regenerated they are likely to contain thousands of meaningless changes, which makes visual inspection of the "diff" between two revisions very difficult.

This example demonstrates some of the weaknesses of a configure stage (because they generate output from a single shell script, they *cannot* be parallelized; they can generate excessive output; common implementations ignore the standard and reimplement more than is necessary on a modern system).

If Autotools is so slow, and so poorly designed, why is it so popular? This is partly due to historical reasons; Autoconf dates to the 1980s, before the creation of POSIX and during a period of great diversity in Unix systems. At that time, Autotools allowed software authors to easily target many of these systems, although it has resulted in an unfortunate culture where code is not often designed to be inherently portable.[12]

In more recent times, the main justification for using Autoconf is its wide variety of build-time features: it easily supports cross-compilers, and can generate build output in user-specified locations if desired. But as a counterpoint, these are not inherent to *how* Autoconf is implemented. Many projects write their own configure scripts that duplicate these useful parts of the Autoconf interface while avoiding the bloat and performance issues of an Autoconf-derived configure script. For example, Musl,[20] an implementation of the C standard library, contains a configure script that aside from generating build logic also provides an Autoconf-compatible interface for cross-compilation, out-of-tree builds, and customization of install location, all in under eight hundred lines. (Contrast with the GNU C library,[21] Musl's main competitor, whose Autoconf-generated configure script is over eight *thousand* lines long.)

2.4 Other design considerations

2.4.1 Declarative Makefiles as user interface

An example of a build system that is primarily declarative is OpenBSD's `bsd.prog.mk`,[22] a set of build rules that is included by the Makefiles of most programs in OpenBSD's source tree.

An OpenBSD installation contains hundreds of executables out of the box. Most of these are built in the exact same way: compiling a collection of source files into an executable, linking against a collection of libraries, and giving the executable a unique name. Including the build logic in each program's makefile is unnecessary duplication; it is better to refactor the identical build logic to a separate include file.

The result is a simple list of variables:

```
PROG=   ls
```

[20] https://musl-libc.org/
[21] https://www.gnu.org/s/libc/
[22] In the OpenBSD 5.8 source tree, visible at `share/mk/bsd.prog.mk`.

```
SRCS=   cmp.c ls.c main.c print.c util.c utf8.c
DPADD= ${LIBUTIL}
LDADD= -lutil

.include <bsd.prog.mk>
```

A typical program needs only specify the program name, which source files to compile, which libraries to link against, as well as adding it as a `make` dependency.

Although `bsd.prog.mk` is not portable, the concept of declarative Makefiles is. The primary benefit of such a Makefile is deduplication. Reducing the amount of build logic means less chance of human error simply by virtue of requiring less human maintenance. In addition, by making the primary interface a simple list of variables, all the capabilities available to the end user are visible in plain sight, removing the user's need to mentally envision invisible data structures when inspecting the Makefile. In these ways, declarative programming is an improvement to the user interface.

2.4.2 Common non-portable `make` constructs

Strict portability is a tradeoff. Sometimes it necessitates giving up useful features. Although equivalent functionality may in some cases be possible through portable means, finding such other methods is a time investment that may provide a diminishing return.

The following subsections discuss two nonstandard `make` features that are commonly supported, widely used, but not portable, and were thus avoided for this project.

Metarules

Metarules, or pattern rules, are a non-portable GNU `make` extension. They act as a general solution for adding new substitution rules and suffix transformations to `make` build logic. A typical example of the shorthand metarules provide is that the new rule `%.o: %.c` would be equivalent to the POSIX-compliant rule `.c.o:`.

POSIX does not require that any version of `make` support metarules, but it does acknowledge their existence as a common extension, and strongly recommends that implementors use this syntax solely for this feature.

Macro shell assignment

Often developers using `make` wish to dynamically assign the output of shell commands to a `make` variable.

For example, previous sections of this thesis have shown examples of makefiles where the user declares the source files or object files used to construct the `make` directed acyclic graphs. A makefile that executes "`ls *.c`" in the shell would generate a list of all files in the current directory dynamically, with no need for the author to manually specify the files that make up the project.

There is some question whether automatically generating filename lists is helpful; after all, it's common while developing to keep temporary source files in the working directory that may not need to be linked to the final result. Nonetheless, collecting filenames with `ls` is only one of the possible uses for macro shell assignment, which having access to the full capabilities of the shell has an almost limitless number of other uses.

As this functionality has never been standardized, different versions of `make` implement it differently, if they implement it at all. The most common is the `!=`

operator. Demonstrating through the previous example, `SRCS != ls *.c` would assign all `.c` filenames to the variable `SRCS`. `!=` is available in OpenBSD `make`, versions of GNU `make` newer than 4.0, and many other `make`s. David A. Wheeler has proposed `!=` to the Austin Group for inclusion in the POSIX standard, but as of the time of this writing it has not yet been accepted by the committee.[19]

2.5 Summary

This chapter explains the rationale behind several design principles appropriate for build systems:

Rather than attempting to support every system equally, build systems should target a single standard system (POSIX), and reimplement standards-compliant capabilities for otherwise incapable systems.

Build systems should make judicious use of interfaces provided by the standard (`make` and the C standard library), in order to avoid dependencies.

They should avoid manual user interaction, such as configure stages, because they are error-prone and require user training and documentation.

A system designed this way eschews complexity. Optimizing for a simple, lightweight infrastructure allows it to be studied, understood and modified quickly and easily by new faces.

Chapter 3

Build system implementation

The previous chapter described features and misfeatures of various build systems. The intent was to learn from them, and apply the lessons to a new project.

This chapter shows the results of last chapter's analysis and demonstrates how the principles map to real code.

3.1 Modular design for additional paradigms

This section introduces a new feature of the proposed build system: software modules that can be seamlessly swapped with alternatively implemented modules that have equivalent functionality.

Consider a hypothetical software tool under development that performs some computation-heavy data processing task. The developers of such a tool might consider alternative hardware platforms, such as GPUs or field-programmable gate arrays, for potential performance increases over a typical CPU. Ideally, they would design their tool in such a way that they could seamlessly replace individual functions written for a CPU with a GPU equivalent, verify that the new module provides correct results, and benchmark the modules to determine what decisions they should make in their final product.

The build system in this chapter will be designed to facilitate such software by providing a simple, clear framework where "alternative" modules keep the same filename as their "original," but are stored in a subdirectory based on which platform the modules run on. These platforms and their associated build infrastructure are called 'paradigms' here.

The build system will include a paradigm that builds modules against CUDA, Nvidia Corporation's parallel programming suite designed to run on GPUs.

3.2 `make` build logic

3.2.1 Overriding POSIX default rules

POSIX specifies a number of default rules for building C and Fortran objects. The "`.c` → `.o`" rule, after variable expansion, is equivalent to:

```
.c.o:
        c99 -O -c $<
```

In other words, when asked to generate a `.o` file, `make` will invoke the C compiler (`c99`) with the default optimization level (`-O`), and generate an object file instead of a linked executable (`-c`).

A rule to generate a fully linked executable from a `.c` file also exists:

```
.c:
        c99 -O -o $@ $<
```

These rules are useful in certain situations. For example, given a C program consisting of a single source file such as `foo.c`, it is trivial to compile the program without needing any makefile at all (by running “`make foo`” from the shell).

Though useful, these rules mostly benefit special cases. In fact, the `.c.o` rule actually causes negative effects in our situation, where programs are assumed to consist of several objects, some sharing filenames, scattered across multiple subdirectories. This is due to the behavior of the C compiler's `-c` flag.

During normal operation, the C compiler generates a linked executable from source files and object files provided as input. When `-c` is passed to the compiler, the result of compilation is an intermediate object file rather than a linked executable.[10]

This turns out to be problematic due to the compiler's default filenames for output files. The GNU C compiler manual describes the naming behavior of `-c` as follows: “By default, the object file name for a source file is made by replacing the suffix `.c`, `.i`, `.s`, etc., with `.o`.” That is true. But what *directory* is the resulting object file stored in? It turns out that it is created in the *current* directory, even if the original source file was located elsewhere.

To see why this is a problem, consider the module system described earlier in this chapter. Individual functions are implemented by individual source files. If a function is written against CUDA, its source code can be found in a subdirectory named `cuda/`; functions written in purely CPU-interpreted code will be located in the same directory as the Makefile. These functions may implement identical interfaces, and may have similar filenames.

If these functions have similar filenames (such as `fft.c` and `cuda/fft.cu`), then the POSIX rule that invokes the C compiler would generate a file called `fft.o`, and a similar rule that invokes the CUDA compiler would generate a file called `fft.o`... in the same directory, resulting in a collision—one file would overwrite the other.

This race condition is a direct result of the compiler flags used by the POSIX definition of the `.c.o` rule. Therefore, we do not use it, but instead define our own replacement. In the case of the C compiler, the POSIX rule:

```
.c.o:
        c99 -O -c $<
```

becomes:

```
.c.o:
        c99 -O -c -o $*.o $<
```

This new rule uses the C compiler's `-o` flag to force the final filename to a given value. The Makefile rule generates the argument to `-o` from the source file's pathname via `make`'s `$*` variable, which strips the `.c` suffix but retains any leading directories. If two modules share a filename but have different suffixes, the object files generated from them cannot overlap, because the design of the build system ensures that files of different suffixes must be kept in separate subdirectories.

3.3 Refactoring build logic

Inspired by Chapter 2's example of `bsd.prog.mk` as a declarative build system, we start with a makefile that compiles both C and CUDA source files, and then factor the build logic into its own makefile so the user-facing makefile can remain primarily declarative.

```
.POSIX:

.SUFFIXES: .c .o .cu

.c.o:
$(CC) $(CFLAGS) -c $< -o $@

.cu.o:
$(NVCC) $(NVCFLAGS) -c $< -o $@
```

In the beginning, we define the special targets `.POSIX` (required by the standard) and `.SUFFIXES` to allow defining inference rules for CUDA files. Next, we create inference rules for both C (replacing the default POSIX inference rules for C) and CUDA.

Although `make` sets `CC` and `CFLAGS` to sensible values by default, `NVCC` and `NVCFLAGS` are custom variables and must be set manually.

```
CUDADIR =
NVCC = /usr/local/cuda-7.5/bin/nvcc
NVCFLAGS = $(CFLAGS)

OBJECTS = foo.o
CUDA_OBJECTS = cuda/foo.o
```

The C and CUDA toolchains behave very similarly, so it makes sense to intentionally mimic the typical C variables and rules when building CUDA code.

```
all: foo cudafoo
```

```
foo: $(OBJECTS)
$(CC) $(CFLAGS) -o $@ $(OBJECTS) -lm

cudafoo: $(CUDA_OBJECTS)
$(NVCC) $(NVCFLAGS) -o $@ $(CUDA_OBJECTS) -lm -lcufft

clean:
        rm -f foo cudafoo
        rm -f $(OBJECTS) $(CUDA_OBJECTS)
```

The exact format of the build rules for the executables themselves depends heavily on the nature of the code being compiled. Any library dependencies will be per project and based on the source files, so it makes sense to reference a variable instead of including library names in the literal rules.

A revised version of the last makefile fragment, then, is as follows:

```
PROGRAM = foo
CLIBS = -lm
CUDALIBS = -lm -lcufft

all: $(PROGRAM) cuda$(PROGRAM)

$(PROGRAM): $(OBJECTS)
        $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(CLIBS)

cuda$(PROGRAM): $(CUDA_OBJECTS)
        $(NVCC) $(NVCFLAGS) -o $@ $(CUDA_OBJECTS) $(CUDALIBS)
```

```
clean:
        rm -f $(PROGRAM) cuda$(PROGRAM)
        rm -f $(OBJECTS) $(CUDA_OBJECTS)
```

The preceding makefile fragments, which are really a single makefile, can be split into a series of variable assignments in one file and the remainder of the logic in a second file which is included by the first using `make`'s `include` keyword.

Build logic for additional paradigms can easily be added to this library makefile. Note that the makefile shown above builds only C and CUDA sources; for reasons discussed in Chapter 4, the FPGA paradigm differs dramatically from C and CUDA and is left out of the generic implementation.

3.4 Software modules

The characteristics of the build system affect the design of the software itself in certain ways. Because the goal is to abstract hardware and software into interchangeable modules, choices in the software API must be considered carefully.

One consequence of the modularity is that memory allocation has to be performed within the modules, not outside of them. This is a casualty of the hardware-generic aspirations of this project; it results from the fact that GPU memory and host memory cannot freely interact with each other. Under normal circumstances, memory would be managed by passing pointers to buffers between functions. Since the GPU/host dichotomy makes that impossible, modules that need to pass data to the outside world do so instead by using static buffers, which are allocated once and then reused anytime the parent function is called. This does mean that benchmarks of an individual module should be discarded for the first run.

It also means the modularity of this approach may be unsuitable for libraries. Memory allocation in C libraries is best performed by a thin wrapper around the `malloc()` function. To see why, we must look at `malloc()`'s definition:

```
void *malloc(size_t size);
```

`malloc()` attempts to allocate `size` bytes of memory, and returns a pointer to the memory if successful. If an error occurs that causes the allocation to fail, the only way to know is by comparing the returned pointer to the `NULL` pointer. But what should happen then?

In a normal application, proper response to a memory allocation failure varies. Some programs may abort. Others may print a diagnostic message.

But if the allocation occurs in a library, what is the proper response? Aborting the program is clearly the wrong thing to do—after all, what if the application wishes to print a diagnostic message? (Imagine a web browser that runs out of memory, then asks the user to close tabs.) Instead, the library should pass the memory allocation to the caller, and the simplest way to do that is by implementing a thin wrapper around `malloc()`.

Since the modules described earlier allocate memory internally without ever exposing the buffer to the outside world, it would be inappropriate to use this module system to implement library code. Once we make the assumption that the modules never contain library code, we even further eliminate the need for complicated error handling. This allows us to optionally write our test programs so that they exit on error. Of course, a program that requires guaranteed reliability in error handling may indeed want to make different decisions such as propagating errors to calling functions. This style of program design, while more complicated than simple error on exit, is more robust.

3.5 Test modules

3.5.1 POSIX clocks

Test modules have two purposes. One is to verify the *correctness* of alternative implementations of the same module through regression testing. This is effectively realized by the concept of small, self-contained functional modules; this build system does not enforce any particular data testing scheme beyond that.

The second purpose is to *benchmark* alternative implementations of a particular module by measuring the time of execution. Although there exists great diversity in timing hardware and operating system clocks, POSIX provides a standard method—indeed, multiple methods—to interface with time.

The three clocks provided by the POSIX standard are a real-time clock, a monotonically increasing clock, and CPU time.[10] The real-time clock is not appropriate for benchmarking as it's subject to skew; the operating system can set the real-time clock at any time, such as when synchronizing with an atomic clock using the Network Time Protocol.

CPU time can be useful in certain situations, but it is useless for benchmarking GPU code as it cannot detect any of the work being done on external hardware.

In contrast, the monotonic clock is quite appropriate for any benchmarking situation. It accurately measures the distance between any two measured points in time. This is often implemented with an interrupt that triggers after a fixed interval of time, during which monotonic clock is incremented. It is not affected by CPU activity or by any skew when the operating system sets time.

Although the monotonic clock is not *required* by the standard, it may be provided as a standard extension and can be detected by the existence of an error when calling the `clock_gettime()` function. Thus, the test suite is capable of providing meaningful error output if the monotonic clock is not available, rather than returning incorrect results at runtime.

3.5.2 Benchmarking module

In order to measure time with the POSIX monotonic clock, two measurements must be taken, and the resulting time difference converted to appropriate units. The benchmarking modules to be discussed in Chapter 4 do so simply and effectly by simply setting two time "marks" via functions provided by a separate benchmarking module.

When using this module, the goal is to measure elapsed time, but it is important to measure the *right* time. As described in Section 3.4, the startup cost of initial memory allocation is significant enough to taint the benchmark results, which are intended to measure recurring costs of the software such as CPU time. A real-world run may last for hours (due either to huge datasets or streaming input) so startup cost gets lost in the noise; but benchmarks and particularly accuracy tests generally aren't expected to take as long, so it's important to make sure this isn't mistakenly included. The implementation in this thesis does so by printing benchmark results in real-time, but discarding the first run entirely.

Chapter 4

Case study: signal recovery

4.1 Introduction

A build system is not very useful without a project to build. This chapter introduces a signal recovery algorithm, notes about its implementation in software and hardware, and considerations regarding its ability to be built and benchmarked with the system described in Chapter 2.

Signal detection theory attempts to separate signal from noise so that information can be extracted. Popular applications include radars, communications, etc. The following implementation uses a "computationally efficient detection" algorithm, or CED, developed by Sandia National Laboratories.

The intent is to benchmark the performance (in execution time). Hence the system was developed in a modular fashion, using the build system to verify consistency in each module's outputs and locating performance bottlenecks.

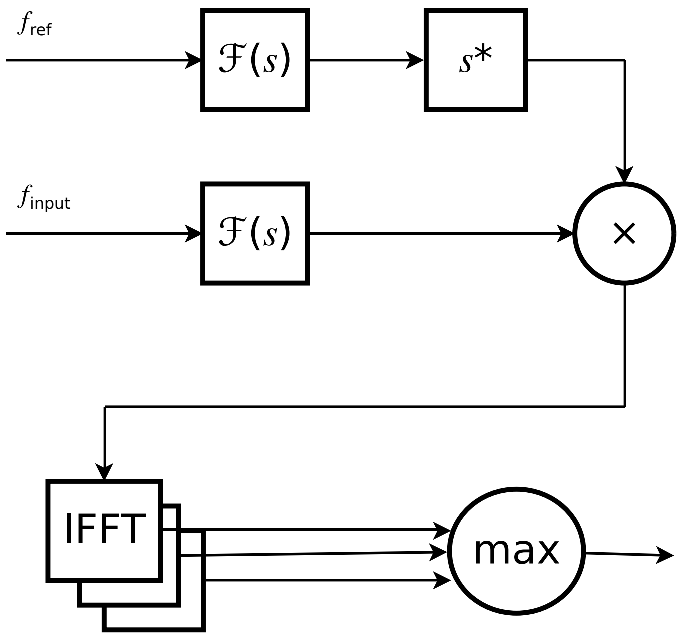

Figure 4.1: Block diagram of the ‘computationally efficient detection’ algorithm.

4.2 The ‘Computationally Efficient Detection’ algorithm

The inputs to the algorithm are a reference signal and a constant stream of input data. The input is repeatedly shifted and cross-correlated against the reference signal; the outputs are an offset into the range corresponding to the highest cross-correlation, and the maximum cross-correlation itself.

The highest cross-correlation generally occurs at the point where the signals are in phase, even if they are not exact. For example, the cross-correlation of a sine wave and a triangle wave is highest when the peaks and valleys of the triangle wave are colocated with the peaks and valleys of the sine; as the two signals go out of phase the cross-correlation will decrease.

Cross-correlation is most often defined as an integral:

$$(f \star g)(\tau) = \int_{-\infty}^{+\infty} f^*(t)\ g(t+\tau)\,dt \tag{4.1}$$

However, it is more useful to define it in terms of convolution:

$$(f \star g) = f^*(-t) * g \tag{4.2}$$

Convolution can be equivalently performed using Fourier transforms:

$$\mathcal{F}\{f * g\} = k \cdot \mathcal{F}\{f\} \cdot \mathcal{F}\{g\} \tag{4.3}$$

Therefore, the cross-correlation can be calculated by Fourier transforms as well:

$$f \star g = \mathcal{F}^{-1}\left\{\mathcal{F}\{f^*(-t)\} \cdot \mathcal{F}\{g\}\right\} \tag{4.4}$$

4.2.1 Translating into code

Pseudocode of the algorithm follows:

```
for (i = 0; i < 3 * reflen; i++) {
        memcpy(buffer, input, inputlen * sizeof(double complex));
        ft(buffer, inputlen);
        for (j = 0; j < inputlen; j++) {
                buffer[j] = conj(buffer[j]) * (ref + i)[j];
        }
        ift(buffer, inputlen);
        d = maxrange(buffer, inputlen);
        if (d > max) {
                max = d;
                maxindex = i;
        }
}
```

The naïve algorithm performs many unnecessary copies. It can be reordered to perform a minimal number of data copies and avoid repeating transforms.

```
memcpy(buffer, input, inputlen * sizeof(double complex));
ft(buffer, inputlen);
for (i = 0; i < inputlen; i++) {
        conjfts[i] = conj(buffer[i]);
}
for (i = 0; i < 3 * reflen; i++) {
        for (j = 0; j < inputlen; j++) {
                buffer[j] = conjfts[j] * (ref + i)[j];
        }
        ift(buffer, inputlen);
        d = maxrange(buffer, inputlen);
        if (d > max) {
                max = d;
                maxindex = i;
        }
}
```

Convolution is a commutative operation, so we can choose to conjugate either the input data or the reference signal. The latter is a better choice, because when the system is used as intended, the input data will dwarf the reference signal in size. Conjugating the input would provide a small but constant performance hit throughout the run of the program; conversely, conjugation of the reference signal only needs to happen once, and the operation falls under the initial startup costs of the software, not affecting the general run time.

4.3 Selection of hardware for implementation

Aside from designing a project that suitably exercises the capabilities of the build system and benchmark system, the motivating factors were to come up with an appropriate 'real-world' system: to solve a specific engineering problem, so that the end result will have some basis in reality.

The CPU is an obvious target: CPUs are inexpensive, extremely common, sufficiently powerful, and widely understood. Doing the work in software allows for quick prototyping and instant portability to a wide variety of platforms, including embedded systems and powerful desktop hardware or even supercomputers. C was picked over C++ due to ubiquity of C compilers combined with the lack of a real need for typical C++ features such as object orientation, templates, data structures provided by the C++ standard library, etc., as described in greater detail in Chapter 2.

Using a GPU to perform calculations is another solid choice. GPUs are off-the-shelf, generic, general-purpose hardware, and have a reputation of substantial performance improvements in calculation-heavy applications.

FPGAs are fairly domain-specific. They are extremely versatile, and open up the possibility of completely new designs that are impossible with the limited (but efficient) capabilities of CPUs and GPUs. There can be potential for great performance due to their flexibility.

4.4 Software implementation

The CPU implementation was written in a modular way. Later, the GPU implementation was done one module at a time.

The software consists of a single process and a single thread of execution. Multithreading was briefly considered but not implemented due to lack of a significant expected benefit versus cost. In order for the software to take advantage of multiple CPUs or multiple cores, it would require either a multiprocess design or the use of one of the standardized threading libraries, POSIX pthreads or ISO C's own threading library. But such an implementation would greatly complicate the code, and yet the benefit, while possibly significant compared to single-core execution, would pale in comparison to the performance improvements demonstrated by a GPU or FPGA implementation of the 'computationally efficient detection' algorithm.

By design, mathematical operations are performed by C's standard math library as much as possible. In particular, complex mathematics are performed using C's native complex API, which is provided by the header `complex.h`, rather than by rewriting the same functionality (e.g., calculating the norm of a complex number with `cabs(x + I * y)` rather than `sqrt(x * x + y * y)`).

`complex.h` defines complex types, `double complex` and `float complex`, each of which consists of a pair of (double- and single-precision, respectively) floating-point numbers representing real and imaginary components.

Complex operations can be performed in a natural manner:

```
double complex c = 2.5 + 7.7 * I;
```

`complex.h` also exposes a number of functions for common complex calculations, such as `cabs()`, which calculates the absolute value more concisely than `sqrt(creal(c) * creal(c) + cimag(c) * cimag(c))`.

The required fast Fourier transforms were performed by the FFTW library. FFTW is a widely used open source library released under the GNU General Public License.

FFTW provides two types for performing complex mathematics. `fftw_complex` is a pair of double-precision floating point numbers, one of which is real and one of which is imaginary. Similarly, `fftwf_complex` is a pair of *single*-precision floating point numbers. Happily, FFTW's complex types are identical to the C standard definitions of `double complex` and `float complex`: an array of one type can be passed seamlessly to a function expecting an array of the other.[3]

Making use of ISO C's _Generic selections to perform type-generic math was considered. Unfortunately, _Generic support is not yet widespread in deployed open source compilers. (In particular, Red Hat Enterprise Linux version 6, the operating system used for both CPU and GPU testing, has as its system compiler GCC version 4.4.7, released in March of 2012.) Instead, measurement of the performance difference between single- and double-precision floating point was performed by the low-tech method of automatically replacing `double` with `float` in the source code.

4.5 GPU Implementation

Nvidia hardware was available, so CUDA was a natural choice for GPU processing. Due to the modularity of the CPU design and the similarity of the Nvidia toolchain to a C compiler—`nvcc` is configured to call an existing compiler driver to do much of the work—it was easy to rewrite the algorithm using GPU capabilities.

Two components of the algorithm lend themselves to parallelism: the Fourier transforms and finding the maximum value of an output array.

4.5.1 Parallelizing Fourier transforms: cuFFT and cuFFTW

In the CPU implementation, the Fourier transforms necessary to find the correlation were calculated by the FFTW library. Likewise, in the GPU implementation, the Fourier transforms are performed by the NVIDIA CUDA Fast Fourier Transform library (cuFFT)[1].

Nvidia actually provides two Fourier transform libraries in the CUDA Software Development Kit: cuFFT and cuFFTW. The latter is a wrapper around cuFFT that provides an interface mimicking FFTW.[17] cuFFTW is intended to be used in software projects that utilize FFTW so they can be easily converted to perform operations on Nvidia GPUs. Although the software implementation of the 'computationally efficient detection' algorithm might fall into this category, its dependency on FFTW is not deeply entangled. Due to the modular design, only a single file, `fft.c`, had to be written in order to use CUDA's native FFT library.

4.5.2 Parallelizing `maxrange.cu`: reduction

The initial version of the code to find a maximum across a range attempted to prevent race conditions by using mutexes. Mutexes, or **mut**ual **ex**clusions, are a synchronization primitive in concurrent execution. When multiple threads are executing a sequence of code that requires a mutex, each will attempt to atomically "grab" the mutex, but only the first to grab it can succeed; the remaining threads block until the mutex is freed.

Using mutexes, the algorithm was essentially a straight copy of the CPU implementation, and defined mutexes (with the CUDA-provided check-and-set function `AtomicCAS()`) to create a critical section in which to perform the max value com-

[1] https://developer.nvidia.com/cufft

parison.

Although this approach makes sense at first to someone used to software development, it is inappropriate for a GPU target. In CUDA, mutexes can only be used per-warp. Using a mutex is fine as long as such use is restricted to a single thread within a warp. But if more than one thread attempts to grab the mutex, the code will hang.

The act of finding the maximum must be represented instead as a parallel reduction problem.

In parallel reduction, the data to be manipulated is broken up into chunks that can be operated on repeatedly by the same process. Once everything has been completed, the results calculated from each chunk are combined into a single dataset, and the same operation is performed again. For very large datasets, the dataset containing the initial results can itself be broken into chunks and operated on in parallel.

Finding the maximum is a natural fit for this kind of algorithm. The maximum value in a set is equal to the maximum of local maxima of subsets, as long as the subsets contain among them contain the entirety of the original set.

In other words, if A is a set, and $B \cup C$ is a proper subset of A (i.e., there are no elements of A that are not also elements of $B \cup C$), then $\max(A) = \max(\{\max(B), \max(C)\})$. Therefore, if $\max(B)$ and $\max(C)$ can be calculated in parallel, there is a clear performance win.

4.5.3 Reduction algorithm optimization

The initial version of the function looked like this:

```
extern __shared__ double maxima[];
```

```
        unsigned int tid = threadIdx.x;
        unsigned int i = blockIdx.x * blockDim.x +
            threadIdx.x > len ? len :
            blockIdx.x * blockDim.x + threadIdx.x;
        maxima[tid] = cuCabs(indata[i]);
        __syncthreads();

        for (unsigned int s = 1; s < blockDim.x; s *= 2) {
                if (tid % (2*s) == 0) {
                        maxima[tid] = fmax(maxima[tid],
                        maxima[tid + s]);
                }

                __syncthreads();
        }
```

Nvidia has published methods for optimizing similar reduction code that calculates sums instead of maxima.[8] Applying the same principles to `maxrange.cu` yields the following:

Replace divergent branching with strided index

The % in the `if` statement detrimental to performance; it causes the warp to diverge (that is, it causes each contiguous group of parallel computations to be performed multiple times to match all possible outcomes). Replace the segment:

```
                if (tid % (2*s) == 0) {
                        maxima[tid] = fmax(maxima[tid],
                        maxima[tid + s]);
                }
```

with:

```
                if (tid < s) {
                        maxima[tid] = fmax(maxima[tid],
                        maxima[tid + s]);
                }
```

Perform some calculations during load

Halve the number of blocks, and then perform an initial maximum calculation while initializing the array:

```
        unsigned int i = blockIdx.x * blockDim.x +
            threadIdx.x > len ? len :
            blockIdx.x * blockDim.x + threadIdx.x;
        maxima[tid] = cuCabs(indata[i]);
```

becomes:

```
        unsigned int i = blockIdx.x * blockDim.x * 2 +
            threadIdx.x > len ? len :
            blockIdx.x * blockDim.x * 2 + threadIdx.x;
        maxima[tid] = fmax(cuCabs(indata[i]),
            cuCabs(indata[i + blockDim.x]));
```

Loop unrolling

```
if (tid == 0) {
        outdata[blockIdx.x] = maxima[0];
}
```

becomes:

```
if (tid < 32) {
        maxima[tid] = fmax(maxima[tid],
            maxima[tid + 32]);
        maxima[tid] = fmax(maxima[tid],
            maxima[tid + 16]);
        maxima[tid] = fmax(maxima[tid],
            maxima[tid + 8]);
        maxima[tid] = fmax(maxima[tid],
            maxima[tid + 4]);
        maxima[tid] = fmax(maxima[tid],
            maxima[tid + 2]);
        maxima[tid] = fmax(maxima[tid],
            maxima[tid + 1]);
        outdata[blockIdx.x] = maxima[0];
}
```

4.5.4 Performance considerations: precision

Should the GPU code calculate using single-precision or double-precision floating point?

The primary market for GPUs has historically been for video games, where a minor inaccuracy due to low precision lasts for but a blink of an eye. Although newer CUDA hardware is capable of high-precision calculations and occasionally introduces added features and improvements for doing so,[21] Nvidia's GPU hardware is generally faster at performing single-precision than double-precision calculations.[1][2]

Similarly, the computationally efficient detection software's use case doesn't require high precision or exact calculations. Using `float` instead of `double` is a sensible design decision.

4.6 FPGA Implementation

While CUDA code is easy to integrate with a C program due to the inherent similarities that come from reusing a C++ compiler driver, FPGA designs are not so similar. The fundamental differences between FPGAs and software programs, as well as significant differences in tooling, make designing hardware modules in a compatible fashion much more challenging.

To integrate with the rest of the system, some simulation is necessary—a design instantiated entirely in hardware will have no way to interface with the other modules of the system. Executing the entire algorithm in simulation is impractical, however, for performance reasons. Simulating Fourier transforms of thousands or millions of points is several orders of magnitude slower than performing the same operations purely in hardware.

Therefore we simulate the outer components, but offload the internal calculations at runtime to the FPGA.[4]

An alternate approach might use a custom wrapper module on the FPGA instead of a testbench. It will be much faster even with the hardware co-simulation. But in turn it will not be as easy to verify, and it requires a great deal more code. E.g., can be done perhaps by dropping in a USB core and communicating with a software helper program; but this requires a dramatic amount of additional software to be written per module (the helper program) and also requires additional hardware design and resources (the USB core).

The simulation portion cannot be easily integrated with the rest of the design. Currently this version simply mimics the interface of the regression tests; an ideal implementation would integrate more closely with the remaining software.

The need for a method to connect software interfaces to a VHDL design was described in a paper from Cadence.[14] Their implementation, VHPI (VHDL Procedural Interface),[6] has been incorporated into the 2008 VHDL standard.[9] However, it is not particularly widespread among vendors yet—particularly Xilinx, who at the time of this writing does not support VHPI at all in their products.

4.7 Test results

The benchmarks list both "setup time"—that is, the time spent preparing to execute the first loop such as reading input files and allocating memory—and "loop time"—the time spent actually performing calculations.

The GPU takes a dramatic amount of time to prepare each function: several *seconds*, while the CPU implementations take mere microseconds. But during the data processing loop, the benchmarks clearly favor the GPU. After 5,000 points in `fft` or 8,000 points in `maxrange`, the GPU implementation comes out ahead, taking less than one third of the time to process 2^{20} points.

This benchmark can influence future decisions. It is clearly advantageous to use the GPU in a streaming fashion, repeatedly processing blocks of thousands or millions of points as they're recorded; the setup time becomes irrelevant while the cumulative benefit of reduced loop time increases with each iteration.

What about `float` versus `double`? In `fft.c`, single-precision loop times are slightly faster than double-precision, ranging from a 6% to 20% improvement, a maximum savings of about 4 ms per loop. In `maxrange.c`, single-precision calculations are invariably 15–17% faster than double-precision. In the GPU modules, single-precision calculations are consistently 44%–55% faster. Using single-precision `float` provides a noticeable improvement in every case.

The various reduction optimizations applied to `maxrange.cu` make little difference. Performance improves by about 0.5% to 3%, or between 10 and 20 microseconds. Because the optimizations require only small changes to the code, they are probably worthwhile, but perhaps time would be better spent searching for other bottlenecks.

module	precision	data length	setup time (μs)	loop time (μs)
fft.c	single	2^{10}	370	24
		2^{11}	299	38
		2^{12}	332	78
		2^{13}	371	195
		2^{14}	484	376
		2^{15}	556	904
		2^{16}	841	1660
		2^{17}	1565	3458
		2^{18}	2807	9098
		2^{19}	3784	21035
		2^{20}	3807	43035
	double	2^{10}	344	24
		2^{11}	336	42
		2^{12}	379	84
		2^{13}	453	208
		2^{14}	651	411
		2^{15}	830	1018
		2^{16}	2528	1853
		2^{17}	4733	4485
		2^{18}	8476	11090
		2^{19}	4967	22334
		2^{20}	4773	47120

module	precision	data length	setup time (μs)	loop time (μs)
`fft.cu`	single	2^{10}	2688954	48
		2^{11}	2675477	51
		2^{12}	2679220	64
		2^{13}	2678584	90
		2^{14}	2676498	143
		2^{15}	2672351	254
		2^{16}	2680241	466
		2^{17}	2679423	911
		2^{18}	2683741	1464
		2^{19}	2690675	2997
		2^{20}	2680750	4830
	double	2^{10}	2681185	59
		2^{11}	2676335	72
		2^{12}	2697671	108
		2^{13}	2696714	152
		2^{14}	2682183	263
		2^{15}	2676765	498
		2^{16}	2685107	985
		2^{17}	2677600	1602
		2^{18}	2676619	2811
		2^{19}	2681900	5261
		2^{20}	2687459	10156

module	precision	data length	setup time (μs)	loop time (μs)
`maxrange.c`	single	2^{10}	9	17
		2^{11}	6	35
		2^{12}	7	70
		2^{13}	17	140
		2^{14}	9	282
		2^{15}	6	569
		2^{16}	19	1129
		2^{17}	25	2273
		2^{18}	43	4512
		2^{19}	27	9037
		2^{20}	253	18078
	double	2^{10}	8	20
		2^{11}	6	40
		2^{12}	5	82
		2^{13}	8	162
		2^{14}	2	331
		2^{15}	10	652
		2^{16}	13	1304
		2^{17}	10	2616
		2^{18}	18	5209
		2^{19}	108	10383
		2^{20}	603	20770

module	precision	data length	setup time (μs)	loop time (μs)
`maxrange.cu`	single	2^{10}	2415461	63
		2^{11}	2390590	67
		2^{12}	2400538	72
		2^{13}	2396537	85
		2^{14}	2395962	110
		2^{15}	2393133	160
		2^{16}	2391298	262
		2^{17}	2405549	615
		2^{18}	2384458	883
		2^{19}	2384378	1404
		2^{20}	2391947	2447
	double	2^{10}	2397766	143
		2^{11}	2383772	147
		2^{12}	2387612	159
		2^{13}	2396757	183
		2^{14}	2393856	271
		2^{15}	2391632	4032
		2^{16}	2371906	4014
		2^{17}	2394944	944
		2^{18}	2404087	1470
		2^{19}	2392970	2515
		2^{20}	2394822	4641

module	precision	data length	setup time (μs)	loop time (μs)
`maxrange.cu` (pre-optimization)	single	2^{10}	2397816	73
		2^{11}	2392495	76
		2^{12}	2386755	81
		2^{13}	2390573	92
		2^{14}	2389357	118
		2^{15}	2393607	171
		2^{16}	2393401	274
		2^{17}	2396502	634
		2^{18}	2388432	889
		2^{19}	2386857	1415
		2^{20}	2395097	2443

Chapter 5

Conclusions

The central thesis of this project has been that portable, standards-compliant software is not difficult to write, does not result in complicated code, and provides tangible maintenance benefit to software engineers.

Chapter 2 developed and justified a set of characteristics of useful build systems: correctness and standards compliance, user interface and automation, and portability across operating systems. It explained the history and proliferation of the C and POSIX standards, along with their advantages and some of their useful features. It then contrasted the simplicity of code written against POSIX to real-world build systems that focus instead on backwards compatibility to old systems that have lost relevance.

Chapter 3 took the beneficial build system characteristics articulated in the previous chapter and used them to create a build system compliant with ISO C and POSIX; it has a clean design that does useful work in a relatively small amount of code. This build system also encourages modular software design. Modules allow a project to be benchmarked or tested for correctness with ease.

Finally, Chapter 4 tests the claims made in previous chapters by implementing a real-world software program that performs signal processing on multiple hardware platforms and using the capabilities of the build system to effectively benchmark the results.

In conclusion, by implementing a project in a modular fashion, a user can effectively benchmark performance of code variants or complete rewrites. This can be done while following standards; portability is not a heavy burden.

Chapter 6

Future Work

This thesis's build system could potentially be simplified by incorporating additional features into the POSIX standard, such as those features described in Section 2.4.2. David A. Wheeler has made several such proposals to the Austin Group; there is ongoing work to push these proposals into the official standard and to implement said features in multiple `make` implementations.[20]

Only a part of the system was implemented on FPGA. It was the version least like the other components. More work needs to be done to determine how to make a HDL description fit this build system, or change the build system to accomodate hardware designs.

A weak point in the FPGA implementation was the tooling for VHDL; software interfaces to VHDL such as Cadence's VHPI have been officially standardized but have not seen wide commercial support. Perhaps Verilog has equivalent features that are more widely implemented; it may be worth investigating how Verilog and VHDL tooling differ.

The CPU implementation was tested with multiple compilers (GCC, LLVM/-

Clang, and PCC). But the GPU implementation was designed exclusively for CUDA. Would it make sense to try this with other GPU frameworks such as OpenCL, which has the advantage of being an open standard?

Multithreading was not seriously considered for this project, as it would provide only a mild increase in CPU performance that would be greatly eclipsed by GPUs' and FPGAs' inherent speed advantages. Nonetheless, projects that might see a serious benefit from multithreading or that for various reasons may not be able to use CUDA or the Xilinx toolchain (for example, due to their proprietary nature) may benefit from such consideration.

Appendices

Appendix A

Build system

Makefile

```
.POSIX:

PROJECT = sigrec

OBJECTS = \
        initarray.o \
        sigrec.o \
        maxrange.o \
        fft.o \
        conjft.o \
        ced.o

CUDA_OBJECTS = \
        initarray.o \
        sigrec.o \
        cuda/maxrange.o \
        cuda/fft.o \
        conjft.o \
        ced.o

CLIBS = -lm -fftwf
CUDALIBS = -lm -fftwf -lcufft

include lib.mk
include test.mk
```

lib.mk

```
CUDADIR = /usr/local/cuda-7.5
CC = cc
NVCC = $(CUDADIR)/bin/nvcc

all: $(PROJECT) cuda$(PROJECT)

CFLAGS += -g -I.
NVCFLAGS = $(CFLAGS)

$(PROJECT): $(OBJECTS)
        $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(CLIBS)

cuda$(PROJECT): $(CUDA_OBJECTS)
        $(NVCC) $(NVCFLAGS) -o $@ $(CUDA_OBJECTS) $(CUDALIBS)
```

```
.SUFFIXES: .c .o .cu

.cu.o:
        $(NVCC) $(NVCFLAGS) -c $< -o $@

clean:
        rm -f $(PROJECT) cuda$(PROJECT)
        rm -f $(OBJECTS) $(CUDA_OBJECTS)
```

test.mk

```
TESTS = tests/testmaxrange tests/cudatestmaxrange

test: $(TESTS)

TESTMAXRANGE_OBJECTS = \
        tests/testmaxrange.o \
        initarray.o \
        test.o \
        maxrange.o

CUDATESTMAXRANGE_OBJECTS = \
        tests/testmaxrange.o \
        initarray.o \
        test.o \
        cuda/maxrange.o

tests/testmaxrange: $(TESTMAXRANGE_OBJECTS)
```

```
        $(CC) -o $@ $(TESTMAXRANGE_OBJECTS) -lm -lrt

tests/cudatestmaxrange: $(CUDATESTMAXRANGE_OBJECTS)
        $(NVCC) -o $@ $(CUDATESTMAXRANGE_OBJECTS) -lm -lrt

TESTFFT_OBJECTS = \
        tests/testfft.o \
        test.o \
        initarray.o \
        fft.o
CUDATESTFFT_OBJECTS = \
        tests/testfft.o \
        initarray.o \
        test.o \
        cuda/fft.o
tests/testfft: $(TESTFFT_OBJECTS)
        $(CC) -o $@ $(TESTFFT_OBJECTS) -lm -lrt -lfftw3f
tests/cudatestfft: $(CUDATESTFFT_OBJECTS)
        $(NVCC) -o $@ $(CUDATESTFFT_OBJECTS) -lm -lrt -lcufft

cleantest: clean
        rm -f tests/*.o
```

Appendix B

Signal recovery implementation

conjft.c

```
#include <tgmath.h>

#include <fftw3.h>

#include "sigrec.h"

/*
 * Given an input signal, find the Fourier transform
 * and conjugate the result.
 */
void
conjft(double complex *v, int len)
{
        int i;
        ft(v, len);
        for (i = 0; i < len; ++i) {
                v[i] = conj(v[i]);
        }
}
```

fft.c

```
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <fftw3.h>

#include "sigrec.h"

/* Take a Fourier transform in-place. */
void
fft(float complex *f, int len, int direction)
{
        fftwf_plan plan;
        static fftwf_complex *tmp;
        if (tmp == NULL) {
                tmp = malloc(len * sizeof(fftwf_complex));
                if (tmp == NULL) {
                        fprintf(stderr, "malloc\n");
                        exit(1);
                }
        }
        memcpy(tmp, f, len * sizeof(fftwf_complex));
        plan = fftwf_plan_dft_1d(len, f, f, direction,
            FFTW_ESTIMATE);
        memcpy(f, tmp, len * sizeof(fftwf_complex));
        if (plan == NULL) {
                fprintf(stderr, "fftw planning failed\n");
                exit(1);
        }
        fftwf_execute(plan);
}
```

initarray.c

```
#include <complex.h>
#include <math.h>
#include <stdio.h>

/*
 * Initialize up to len elements of an array with points
 * read from a file.
 * Return the number of elements initialized, or -1 on error.
 */
int
initarray(float complex *data, FILE *f, int len)
{
        float d, e;
        int i;

        i = 0;
        while (len-- && fscanf(f, "%f %f\n", &d, &e) == 2) {
                data[i++] = d + I * e;
        }

        return i;
}
```

ced.c

```
#include <tgmath.h>

#include <fftw3.h>

#include "sigrec.h"

/*
 * Perform a computationally efficient detection (CED) calculation.
 */

double
CED(double complex *conjfts, double complex *ref,
    int reflen, double complex *scratch, int scratchlen)
{
        int i, j, maxindex;
        double d, max;
        max = 0;
        for (i = 0; i < 3 * reflen; ++i) {
                for (j = 0; j < scratchlen; ++j) {
                        scratch[j] = conjfts[j] * (ref + i)[j];
                }
                ift(scratch, scratchlen);
                d = maxrange(scratch, scratchlen);
                if (d > max) {
                        max = d;
                        maxindex = i;
                }
        }

        return max;
}
```

maxrange.c

```
#include <tgmath.h>

#include <fftw3.h>

#include "sigrec.h"

float
maxrange(float complex *v, int len)
{
        int i;
        float max;
        for (i = 0, max = 0; i < len; ++i) {
                max = fmax(max, fabs(v[i]));
        }

        return max;
}
```

sigrec.c

```
#include <complex.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <fftw3.h>

#include "sigrec.h"

void
printvec(double complex *v, int len, double(*f)(double complex))
{
        int i;
        static int flag;
        if (flag) {
                fprintf(stderr, "called printvec twice\n");
                exit(0);
        }
        for (i = 0; i < len; ++i) {
                printf("%e\n", f(v[i]));
        }
        flag = 1;
}

int
main(int argc, char *argv[])
{
        int len = 1000;
        int inputlen, reflen;
        int i, j;
        double complex *ref;
        double complex *input;
        double complex *tmpinput;
        double complex *inputs;
        double max, d;
        int maxindex;

        if (argc != 3) {
                fprintf(stderr,
                    "usage: sigrec ref.dat input.dat\n");
                exit(1);
        }

        input = (double complex *)fftwf_malloc(
            sizeof(double complex) * len);
        ref = (double complex *)fftwf_malloc(
            sizeof(double complex) * len * 3);
        if (input == NULL || ref == NULL) {
                fprintf(stderr, "memory allocation failed\n");
                exit(1);
        }

        FILE *f = fopen(argv[1], "r");
        if (f == NULL) {
                fprintf(stderr,
                    "could not open file %s: %s\n", argv[1],
                    strerror(errno));
                exit(1);
        }
        FILE *g = fopen(argv[2], "r");
        if (g == NULL) {
                fprintf(stderr,
                    "could not open file %s: %s\n", argv[2],
                    strerror(errno));
                exit(1);
        }
        reflen = initarray(ref + len, f, len);
        if (reflen == 0) {
                fprintf(stderr,
                    "no data in file %s\n", argv[1]);
                exit(1);
        }
        ref = ref + len - reflen;
        inputlen = initarray(input, g, len);
        if (inputlen == 0) {
                fprintf(stderr,
                    "no data in file %s\n", argv[2]);
                exit(1);
        }

        conjft(ref + reflen, reflen);

        memcpy(ref, ref + reflen,
            reflen * sizeof(double complex));
        memcpy(ref + reflen * 2, ref,
            reflen * sizeof(double complex));

        max = 0;
        maxindex = 0;
        tmpinput = malloc(inputlen * sizeof(double complex));
        inputs = malloc(reflen * sizeof(double complex));

        memcpy(inputs, input,
            inputlen * sizeof(double complex));
        ft(inputs, inputlen);
        max = CED(inputs, ref, reflen,
            tmpinput, inputlen);

        return 0;
}
```

test.c

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

void
mark1(struct timespec *t)
{
        if (clock_gettime(CLOCK_MONOTONIC, t) == -1) {
                fprintf(stderr,
                    "mark1:clock_gettime(): %s\n",
                    strerror(errno));
                exit(1);
        }
}

time_t
mark2(struct timespec *t1)
{
        struct timespec t2;
        if (clock_gettime(CLOCK_MONOTONIC, &t2) == -1) {
                fprintf(stderr,
                    "mark2:clock_gettime(): %s\n",
                    strerror(errno));
                exit(1);
        }

        return t2.tv_sec * 1000000 + t2.tv_nsec / 1000 -
             (t1->tv_sec * 1000000 + t1->tv_nsec / 1000);
}
```

testfft.c

```
#include <complex.h>
#include <errno.h>
#include <getopt.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include <fftw3.h>

#include "sigrec.h"
#include "test.h"

int
main(int argc, char *argv[])
{
        struct timespec t;
        float complex *data;
        FILE *f;
        int len = 5000000;
        unsigned long n;
        char *ep;

        int ch;
        while ((ch = getopt(argc, argv, "n:")) != -1) {
                switch (ch) {
                case 'n':
                        errno = 0;
                        n = strtoul(optarg, &ep, 10);
                        if (!(optarg[0] ==  '\0' ||
                            *ep != '\0' ||
                            errno == ERANGE &&
                            n == ULONG_MAX))
                                len = n;
                        break;
                default:
                        return 1;
                }
        }
        argc -= optind;
        argv += optind;

        if (argc != 1) {
                fprintf(stderr,
                    "usage: testfft [-n numpoints]"
                    " input.dat\n");
                return 1;
        }

        f = fopen(argv[0], "r");
        if (f == NULL) {
                fprintf(stderr,
                    "Could not open file %s: %s\n",
                    argv[0],
                    strerror(errno));
                return 1;
        }

        data = malloc(len * sizeof(float complex));
        if (data == NULL) {
                fprintf(stderr, "malloc failure\n");
                return 1;
        }

        len = initarray(data, f, len);

        mark1(&t);
        int m1, m2, m3, m4, m5;

        ft(data, len);
        m1 = mark2(&t);
        ft(data, len);
        m2 = mark2(&t);
        ft(data, len);
        m3 = mark2(&t);
        ft(data, len);
        m4 = mark2(&t);
        ft(data, len);
        m5 = mark2(&t);

        m5 -= m4;
        m4 -= m3;
        m3 -= m2;
        m2 -= m1;
        m1 -= m2;
        printf("%d points;\n"
            "setup time %d us;\n"
            "loop time %d us;\n"
            "mean loop %d us\n", len, m1, m2,
                (m2 + m3 + m4 + m5) / 4);

        return 0;
}
```

testmaxrange.c

```
#include <errno.h>
#include <getopt.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <tgmath.h>
float maxrange(float complex *v, int len);
```

```
int
main(int argc, char *argv[])
{
        struct timespec t;
        float complex *data;
        FILE *f;
        int len = 5000000;
        unsigned long n;
        char *ep;

        int ch;
        while ((ch = getopt(argc, argv, "n:")) != -1) {
                switch(ch) {
                case 'n':
                        errno = 0;
                        n = strtoul(optarg, &ep, 10);
                        if (!(optarg[0] == '\0' ||
                            *ep != '\0' ||
                            errno == ERANGE &&
                            n == ULONG_MAX))
                                len = n;
                        break;
                default:
                        return 1;
                }
        }
        argc -= optind;
        argv += optind;

        if (argc != 1) {
                fprintf(stderr,
                    "usage: testmaxrange "
                    "[-n numpoints] input.dat\n");
                return 1;
        }

        f = fopen(argv[0], "r");
        if (f == NULL) {
                fprintf(stderr,
                    "Could not open file %s: %s\n",
                    argv[0], strerror(errno));
                return 1;
        }

        data = malloc(len * sizeof(float complex));
        if (data == NULL) {
                fprintf(stderr, "malloc failure\n");
                return 1;
        }

        len = initarray(data, f, len);

        mark1(&t);
        int m1, m2, m3, m4, m5;
        maxrange(data, len);
        m1 = mark2(&t);
        maxrange(data, len);
        m2 = mark2(&t);
        maxrange(data, len);
        m3 = mark2(&t);
        maxrange(data, len);
        m4 = mark2(&t);
        maxrange(data, len);
        m5 = mark2(&t);

        m5 -= m4;
        m4 -= m3;
        m3 -= m2;
        m2 -= m1;
        m1 -= m2;

        printf("%d points;\n"
            "setup time %d us;\n"
            "loop time %d us;\n"
            "mean loop %d us\n", len, m1, m2,
                (m2 + m3 + m4 + m5) / 4);

        printf("%d %d %d %d %d\n",
            m1, m2, m3, m4, m5);

        return 0;
}
```

fft.cu

```
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <cuda.h>
#include <cufft.h>
#include <fftw3.h>

//#include "sigrec.h"

/* Take a Fourier transform in-place. */
extern "C" void
fft(float complex *f, int len, int direction)
{
        static cuFloatComplex *tmp;
        cufftResult result;
        static cufftHandle plan;

        if (tmp == NULL) {
                cudaMalloc(&tmp,
                    sizeof(cuFloatComplex) * len);
        }
        cudaMemcpy(tmp, f,
            sizeof(cuFloatComplex) * len,
            cudaMemcpyHostToDevice);
        if (plan == 0) {
                result = cufftPlan1d(&plan, len,
                    CUFFT_C2C, 1);
                if (result != CUFFT_SUCCESS) {
                        fprintf(stderr, "%d ",
                            result);
                        fprintf(stderr,
                            "cufftPlan1d()\n");
                        exit(1);
                }
        }

        cufftExecC2C(plan, tmp, tmp,
            direction == FFTW_FORWARD
            ? CUFFT_FORWARD
            : CUFFT_INVERSE);

        cudaMemcpy(f, tmp,
            sizeof(float complex) * len,
            cudaMemcpyDeviceToHost);
}
```

maxrange.cu

```
#include <complex.h>
#include <math.h>
#include <stdio.h>
```

```
#include <stdlib.h>

#include <cuComplex.h>

#define BLOCK_SIZE 256
#define GRID_SIZE (32 * 8)

__global__ void static
gpumaxrange(cuFloatComplex *indata,
    float *outdata, int len)
{
        extern __shared__ float maxima[];
        unsigned int tid = threadIdx.x;
        unsigned int i = blockIdx.x * blockDim.x * 2
            + threadIdx.x > len ? len :
            blockIdx.x * blockDim.x * 2 +
            threadIdx.x;
        maxima[tid] = fmax(cuCabsf(indata[i]),
            cuCabsf(indata[i + blockDim.x]));
        __syncthreads();

        for (unsigned int s = blockDim.x / 2;
            s > 32; s >>= 1) {
                if (tid < s) {
                        maxima[tid] =
                            fmax(maxima[tid],
                            maxima[tid + s]);
                }
                __syncthreads();
        }

        if (tid < 32) {
                maxima[tid] = fmax(maxima[tid],
                    maxima[tid + 32]);
                maxima[tid] = fmax(maxima[tid],
                    maxima[tid + 16]);
                maxima[tid] = fmax(maxima[tid],
                    maxima[tid + 8]);
                maxima[tid] = fmax(maxima[tid],
                    maxima[tid + 4]);
                maxima[tid] = fmax(maxima[tid],
                    maxima[tid + 2]);
                maxima[tid] = fmax(maxima[tid],
                    maxima[tid + 1]);
                outdata[blockIdx.x] = maxima[0];
        }
}

extern "C" float
maxrange(float complex *v, int len)
{
        float dmax;
        cuFloatComplex *idata_dev;
        cudaMalloc(&idata_dev,
            sizeof(cuFloatComplex) * len);
        cudaMemcpy(idata_dev, v,
            sizeof(cuFloatComplex) * len,
            cudaMemcpyHostToDevice);

        float *odata, *odata_dev;
        cudaMalloc(&odata_dev,
            sizeof(float) * GRID_SIZE);
        odata = (float *)malloc(sizeof(float)
            * GRID_SIZE);

        gpumaxrange <<<GRID_SIZE, BLOCK_SIZE,
            BLOCK_SIZE * sizeof(float)>>>
            (idata_dev, odata_dev, len);
        cudaMemcpy(odata, odata_dev,
            sizeof(float) * GRID_SIZE,
            cudaMemcpyDeviceToHost);

        dmax = -INFINITY;
        for (int i = 0; i < GRID_SIZE; i++) {
                dmax = fmax(dmax, odata[i]);
        }

        return dmax;
}
```

References

[1] NVIDIA's Next Generation CUDA Compute Architecture: Fermi. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[2] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[3] *FFTW User Manual*, 3.2.1 edition, 2009.

[4] UG817: ISim Hardware Co-Simulation Tutorial: Accelerating Floating Point Fast Fourier Transform Simulation. Technical report, Xilinx, 2011.

[5] Adam de Boor. PMake—A Tutorial. https://www.freebsd.org/doc/en/books/pmake/book.html, 1989.

[6] Doub Dunlop. VHPI, A Programming Language Interface for VHDL. Technical report, Cadence Design Systems, Inc., 1996.

[7] Marc Espie. Making Make Parallel. Technical report, The OpenBSD Project, 2014.

[8] Mark Harris. Optimizing Parallel Reduction in CUDA. Technical report, NVIDIA.

[9] Institute of Electrical and Electronics Engineers. *IEEE Standard 1076-2008: VHDL Language Reference Manual*, 2008.

[10] Institute of Electrical and Electronics Engineers. *IEEE Std 1003.1, 2013: IEEE Standard Portable Operating System Interface for Computer Environments*, 2013.

[11] International Organization for Standardization. *ISO/IEC 9899:2011: C international standard*, 2011.

[12] Poul-Henning Kamp. A Generation Lost in the Bazaar. *ACM Queue*, 2012.

[13] Brian Kernighan and Rob Pike. The UNIX Programming Environment, 1984.

[14] Françoise Martinolle, Charles Dawson, Debra Corlette, and Mike Floyd. Interoperability of Verilog/VHDL Procedural Language Interfaces to biuld a mixed language GUI. Technical report, Cadence Design Systems, Inc., 1999.

[15] Damien J. Miller. Secure Portability. Technical report, The OpenBSD Project, 2005.

[16] Peter A. Miller. Recursive Make Considered Harmful. The Australian UNIX and Open Systems Users Group, 1997.

[17] NVIDIA. *CUFFT Library User's Guide*, v7.5 edition, 2015.

[18] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8, 1995.

[19] David A. Wheeler. Add standard support for "!=" (macro shell assignment) in make. `http://austingroupbugs.net/view.php?id=337`, 2010.

[20] David A. Wheeler. Improving Make. `http://www.dwheeler.com/essays/make.html`, 2014.

[21] Nathan Whitehead and Alex Fit-florea. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. `http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf`.