Parallelizing Julia with a Non-invasive DSL

The ECOOP 2017 paper "Parallelizing Julia with a Non-invasive DSL" describes ParallelAccelerator, an embedded DSL and compiler for speeding up compute-intensive Julia programs. In particular, Julia code that makes heavy use of aggregate array operations is a good candidate for speeding up with ParallelAccelerator.

This artifact is a companion to the paper and describes how to install, run, test, and benchmark the ParallelAccelerator Julia package. We will make frequent references to parts of the paper to help explain aspects of the artifact. (Readers may want to skim sections 1-3 of the paper to get an idea of what ParallelAccelerator is about before examining the artifact.)

What's in this artifact

Aside from this README, this artifact has two top-level directories:

- `julia-packages`
- `benchmarking`

Julia packages

ParallelAccelerator has been released as an open-source Julia package, with code available on GitHub. We've also released the CompilerTools package on which ParallelAccelerator depends. This artifact includes snapshots of the most recent tagged releases of both packages (ParallelAccelerator v0.2.2 and CompilerTools v0.2.1) at the time of artifact release. These code snapshots are in the `julia-packages` directory.

It is possible to install the ParallelAccelerator and CompilerTools packages manually, directly from the snapshots in this artifact. However, we recommend instead installing them from the registered package versions on the Internet, using Julia's `Pkg.add()` functionality, following the directions in the below **"Installing and testing ParallelAccelerator"** section. **The code included here is identical to what `Pkg.add()` will download and install at the time of artifact release**; the reason to use `Pkg.add()` is that then Julia will automatically build the packages and install their dependencies for you.

Benchmarking

The empirical evaluation in section 6 of our paper measures the performance of ParallelAccelerator implementations of a suite of ten workloads, compared to plain Julia implementations. The source code for the ParallelAccelerator and plain Julia implementations of those workloads (as well as a few other workloads) is included in this artifact under the directory `julia-packages/ParallelAccelerator.jl-0.2.2/examples` (for the ParallelAccelerator versions) and `julia-packages/ParallelAccelerator.jl-0.2.2/examples/plain-julia` (for the plain Julia versions). These are discussed in the **"How to run the examples"** section, below.

Section 6 of our paper also compares each workload against other implementations – typically MATLAB, but we also have Python or C/C++ implementations of a few workloads. We've included all of the code for the MATLAB, Python, and C/C++ implementations of the workloads in the `benchmarking` directory of this artifact, along with the code we used to do the actual benchmarking

and plotting. We discuss how to use this code in the **"Repeating the experiments from the paper"** section below.

System and software dependencies

The bare minimum you need to use ParallelAccelerator is:

- A *nix OS, ideally Linux.
- A recent installation of Julia.
- A C/C++ compiler, ideally ICC.

To make ParallelAccelerator run to its full potential (that is, to make matrix multiplication go fast), you'll also need:

- A BLAS library, ideally Intel MKL.

More details about each of these dependencies are below.

Operating system

You need a *nix operating system to use ParallelAccelerator. Platforms we have tested on include Ubuntu 16.04, Ubuntu 14.04, CentOS 6.6, macOS Yosemite, and macOS Sierra. **We recommend using Linux.**

Why not macOS? ParallelAccelerator makes use of the C/C++ compiler's OpenMP support. If OpenMP isn't available, you'll see the message `OpenMP is not used.` as each test runs. This is generally not an issue on Linux, but under the current version of ParallelAccelerator, it takes serious effort to fix on macOS and involves modifying ParallelAccelerator itself. Therefore, although ParallelAccelerator will run on macOS, for the time being we recommend Linux.

It may be possible to run ParallelAccelerator on a Windows system using MinGW or the like, but we haven't tested it and don't recommend it.

Installing Julia

To use ParallelAccelerator, you need a recent installation of Julia. **We recommend Julia v0.5.x.** If you don't have an up-to-date version of Julia installed, go to the Julia download page and download the appropriate version for your platform listed under "Current Release".

Check that you can run Julia and get to a `julia>` prompt. You will know you're running the correct version if you see `Version 0.5.x` where x is 0 or greater.

Note about Julia versions: The current Julia release is 0.5.1, which ParallelAccelerator should work fine with. Julia 0.6.0 is due to be released in the very near future, and ParallelAccelerator is *not* guaranteed to work with 0.6.0.

All the benchmarking in section 6 of our paper was done with Julia version 0.5.0, which was the most recent release at the time of paper submission. The exception (as noted in footnote 8 of the paper) is the Horn-Schunck optical flow workload, for which we show results for Julia 0.4.6 (because, for that one workload, a performance regression bug in Julia 0.5.0 caused a large slowdown in the standard Julia implementation that would make the comparison unfair).

ParallelAccelerator should work on Julia 0.4.x, but it will not work with versions of Julia prior to v.0.4.0.

In the rest of this document, we'll assume that you are using Julia 0.5.x and that your Julia packages are installed in `~/.julia/v0.5/` (the default for Julia 0.5.x).

C/C++ compiler

ParallelAccelerator depends on an external C/C++ compiler. This dependency is necessary because ParallelAccelerator works by compiling Julia to C++ using its "CGen" backend, as described in section 5.3 ("Code Generation") of our paper.

C/C++ compilers we have tried ParallelAccelerator with include ICC 17.0.2, ICC 15.0.2, GCC 6.3.0, and GCC 4.8.4.

For best results, we recommend installing ICC, which is available as part of a free 30-day trial of Intel Parallel Studio XE (requires registration). Also included with Intel Parallel Studio XE is Intel MKL, which is the BLAS library with which ParallelAccelerator works best (see the **"BLAS library"** section below).

At package build time, ParallelAccelerator will check to see if you have ICC installed, and if so, it will default to using ICC; otherwise, it will use GCC.

To get Intel Parallel Studio XE, go to this page, select your OS and click "Download FREE Trial", and fill out the form. You'll receive an email with a link that will allow you to download the software. On Linux, download and extract the .tgz file and run the `install.sh` script at the top level of the extracted directory. On macOS, run the downloaded installer. You'll have the option of installing a variety of components, including **Intel C++ Compiler** and **Intel Math Kernel Library (MKL)** for your platform. You should make sure to select both of these components as part of your installation.

After following the ICC download and installation process, you will need to run the `compilervars.sh` script to set environment variables. For most users, the command will be:

```
$ source /opt/intel/bin/compilervars.sh intel64
```

You may also need to do the analogous thing for MKL:

```
$ source /opt/intel/mkl/bin/mklvars.sh intel64
```

BLAS library

If you installed the Intel Math Kernel Library (MKL) along with ICC above, then you already have a BLAS library. If not, then as ParallelAccelerator runs, you'll frequently see the following message:

```
WARNING: MKL and OpenBLAS not found. Matrix multiplication might be slow.
     Please install MKL or OpenBLAS and rebuild ParallelAccelerator for better performance.
```

As an alternative to MKL, we include instructions for installing OpenBLAS on macOS and Ubuntu. Other Linux platforms should be similar to Ubuntu.

After installing OpenBLAS, you should no longer see the `MKL and OpenBLAS not found. Matrix multiplication might be slow.` messages. However, **two of the tests are known to fail** with an error something like this:

```
Undefined symbols for architecture x86_64:
  "_cblas_domatcopy", referenced from:
    ppgemv_t2p1147(j2c_array<double>&, j2c_array<double>&, j2c_array<double>*) in cgen_output69.o
    ppgemv_t2p1147_unaliased(j2c_array<double>&, j2c_array<double>&, j2c_array<double>*) in cgen_outp
```

This is because ParallelAccelerator uses a few features that are supported in MKL but not in OpenBLAS. We therefore recommend using MKL if at all possible.

OpenBLAS on Ubuntu

On Ubuntu, all you should need to do is:

```
$ sudo apt-get install libopenblas-dev
```

If you've already installed ParallelAccelerator at this point, remember to run the following command at the `julia>` prompt to rebuild ParallelAccelerator:

```
julia> Pkg.build("ParallelAccelerator")
```

OpenBLAS on macOS

On macOS, getting set up with OpenBLAS is a bit more involved. Firstly, you can install OpenBLAS via Homebrew:

```
$ brew install homebrew/science/openblas
```

At this point, you should have OpenBLAS installed in `/usr/local/opt/openblas`. You will need to set several environment variables to make ParallelAccelerator aware of it:

```
$ export C_INCLUDE_PATH=/usr/local/opt/openblas/include/
$ export CPLUS_INCLUDE_PATH=/usr/local/opt/openblas/include/
$ export LD_LIBRARY_PATH=/usr/local/opt/openblas/lib
```

If you've already installed ParallelAccelerator at this point, remember to run the following command at the `julia>` prompt to rebuild ParallelAccelerator:

```
julia> Pkg.build("ParallelAccelerator")
```

Installing and testing ParallelAccelerator

Once you've confirmed that you've satisfied the above dependencies, run Julia and run the command `Pkg.add("ParallelAccelerator")` from the `julia>` prompt. This will install the most recent tagged release of ParallelAccelerator and its dependencies. You'll see something like the following:

```
julia> Pkg.add("ParallelAccelerator")
INFO: Initializing package repository /Users/username/.julia/v0.5
INFO: Cloning METADATA from https://github.com/JuliaLang/METADATA.jl
INFO: Installing Compat v0.23.0
INFO: Installing CompilerTools v0.2.1
INFO: Installing DataStructures v0.5.3
INFO: Installing DocOpt v0.2.1
INFO: Installing ParallelAccelerator v0.2.2
INFO: Building ParallelAccelerator
ParallelAccelerator: build.jl begin.
ParallelAccelerator: Building j2c-array shared library
Using icpc to build ParallelAccelerator array runtime.
ParallelAccelerator: build.jl done.
INFO: Package database updated
```

If you're using GCC instead of ICC, you'll see `Using g++ to build ParallelAccelerator array runtime.` instead of `Using icpc to build ParallelAccelerator array runtime.` You may see some compiler warnings that you can safely ignore at this point.

Now you can test the installed package:

```
julia> Pkg.test("ParallelAccelerator")
```

This will first install various dependencies, which may take a while.

If using GCC, you may see some compiler warnings that you can safely ignore. If you see `MKL FATAL ERROR: Cannot load libmkl_avx.so or libmkl_def.so.` on some tests, this is a known issue on some platforms. It may be resolved by running `source /opt/intel/mkl/bin/mklvars.sh intel64`.

How to run the examples

Now that you're set up with ParallelAccelerator, let's see how it performs on your machine on various workloads.

Caveat

ParallelAccelerator tries to exploit the available parallel hardware. The benchmarks in the paper were run on a machine with 36 physical cores and 128 GB of RAM (see section 6 of the paper for complete machine specifications). Running on a machine with fewer cores will naturally result in smaller speedups, but most people don't have such a beefy machine on their desks.

The below examples were run on an 8-core desktop Linux machine with 8 GB of RAM, using ICC and MKL, which is closer to what the average user is likely to have available.

Black-Scholes option pricing benchmark

The directory `~/.julia/v0.5/ParallelAccelerator/examples` contains various example programs showing how to use ParallelAccelerator. Let's start with a classic workload: the Black-Scholes option pricing benchmark. See section 3.1 of our paper for a detailed discussion of this workload, and section 6.2 of the paper for a discussion of its performance.

You can run the Black-Scholes example from the `julia>` prompt:

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
black-scholes/black-scholes.jl")
```

You may see some warnings like the following, which are harmless:

```
WARNING: Method definition cndf2(Any) in module Main at /home/username/.julia/v0.5/ParallelAccelerator/
WARNING: Method definition blackscholes(Any, Any, Any, Any, Any) in module Main at /home/username/.julia
WARNING: Method definition run(Any) in module Main at /home/username/.julia/v0.5/ParallelAccelerator/ex
WARNING: Method definition main() in module Main at /home/username/.julia/v0.5/ParallelAccelerator/exam
```

On your first run of the Black-Scholes examples, you should see output similar to the following (this was on an 8-core Linux machine with 8 GB of RAM, using ICC and MKL):

```
iterations = 10000000
SELFPRIMED 31.675743032
checksum: 2.0954821257116845e8
rate = 2.268548657657791e7 opts/sec
SELFTIMED 0.44081047
```

The SELFTIMED line in the printed output shows the running time, while the SELFPRIMED line shows the time it takes to compile the accelerated code and run it with a small "warm-up" input.

For the first function you accelerate in a given Julia session, you might notice a long time being reported for SELFPRIMED. This delay (about 25 to 30 seconds on an 8-core desktop machine) is the time it takes for Julia to load the ParallelAccelerator package itself. If you run the function again, you'll notice that SELFPRIMED gets smaller. Here's an example interaction:

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
black-scholes/black-scholes.jl")
iterations = 10000000
SELFPRIMED 31.675743032
checksum: 2.0954821257116845e8
rate = 2.268548657657791e7 opts/sec
SELFTIMED 0.44081047

julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
black-scholes/black-scholes.jl")
iterations = 10000000
SELFPRIMED 1.62395378
checksum: 2.0954821257116845e8
rate = 2.3933823208592944e7 opts/sec
SELFTIMED 0.417818746

julia>
```

Notice that SELFPRIMED dropped from almost 32 seconds to about 1.6 seconds. This is because, for the second run, the ParallelAccelerator package was already loaded. The remaining 1.6 seconds is mostly the time it took for the ParallelAccelerator compiler to compile the accelerated code.

It's instructive to compare the running time of the ParallelAccelerator code with the plain Julia version:

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
plain-julia/black-scholes/black-scholes.jl")
iterations = 10000000
SELFPRIMED 0.000573466
checksum: 2.0954821257116845e8
rate = 3.640941015492507e6 opts/sec
SELFTIMED 2.746542709
```

Notice that this time, SELFPRIMED is almost 0 (because there's no package to be loaded and no work for the ParallelAccelerator compiler to do), but SELFTIMED is rather larger than it was under ParallelAccelerator: 2.7 seconds.

You can also run example programs from the command line:

```
$ julia ~/.julia/v0.5/ParallelAccelerator/examples/black-scholes/black-scholes.jl
```

Pass the `--help` option to see usage information for each example:

```
$ julia ~/.julia/v0.5/ParallelAccelerator/examples/ \
black-scholes/black-scholes.jl -- --help
black-scholes.jl

Black-Scholes option pricing model.

Usage:
  black-scholes.jl -h | --help
  black-scholes.jl [--iterations=<iterations>]

Options:
  -h --help                  Show this screen.
  --iterations=<iterations>  Specify a number of iterations [default: 10000000].
```

For this particular workload, the default number of iterations is 10,000,000, but in our paper we benchmark against 100,000,000 iterations. Let's try that. Here's the ParallelAccelerator version, which runs in about 2.3 seconds on our 8-core desktop machine:

```
$ julia ~/.julia/v0.5/ParallelAccelerator/examples/
black-scholes/black-scholes.jl -- --iterations=100000000
iterations = 100000000
SELFPRIMED 30.311812949
checksum: 2.0954821326145482e9
rate = 4.37808254525881e7 opts/sec
SELFTIMED 2.284104947
```

What happens when we run the plain Julia version?

```
$ julia ~/.julia/v0.5/ParallelAccelerator/examples/ \
plain-julia/black-scholes/black-scholes.jl -- --iterations=100000000
iterations = 100000000
SELFPRIMED 2.0588e-5
ERROR: LoadError: OutOfMemoryError()
 in broadcast_t at ./broadcast.jl:228 [inlined]
 in broadcast at ./broadcast.jl:230 [inlined]
[...]
```

The plain Julia version runs out of memory! (For the results in the paper, we ran on a machine with 128 GB of RAM, where this 100,000,000-iteration run took about 22 seconds to run (see section 6.1 of the paper); your mileage may vary.)

More examples

After running the Black-Scholes benchmark, feel free to try any of the other ParallelAccelerator examples. Here are several to try:

Gaussian blur (described in sections 3.2 and 6.3 of the paper):

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
gaussian-blur/gaussian-blur.jl")
```

2D wave equation simulation (described in sections 3.3 and 6.8 of the paper):

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
wave-2d/wave-2d.jl")
```

Horn-Schunck optical flow (described in section 6.1 of the paper):

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
opt-flow/opt-flow.jl")
```

Laplace 3D 6-point stencil (described in section 6.4 of the paper):

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
laplace-3d/laplace-3d.jl")
```

Quantitative option pricing model (described in section 6.5 of the paper):

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
quant/quant.jl")
```

2D lattice Boltzmann fluid flow model (described in section 6.6 of the paper):

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
boltzmann/boltzmann.jl")
```

Harris corner detection (described in section 6.7 of the paper):

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
harris/harris.jl")
```

Julia set computation (Julia sets in Julia! Who could resist?) (described in section 6.9 of the paper):

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
juliaset/juliaset.jl")
```

Nengo Neural Engineering Framework algorithm (described in section 6.10 of the paper):

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
nengo/nengo-nef.jl")
```

For all of these examples, there is a plain Julia version available in the `plain-julia` directory under `examples`. For instance, to run the plain Julia version of Gaussian blur:

```
julia> include("$(homedir())/.julia/v0.5/ParallelAccelerator/examples/ \
plain-julia/gaussian-blur/gaussian-blur.jl")
```

In comparison to the ParallelAccelerator versions, many of the plain Julia versions run quite slowly (and may run out of memory).

Repeating the experiments from the paper

Now we turn to the `benchmarking` directory of the artifact.

The script `benchmarking/util/env-vars.sh` sets various environment variables necessary for benchmarking. Take a look at `env-vars.sh` to make sure that it points to the correct Julia executable on your system. Then, from the artifact directory, run:

```
$ source benchmarking/util/env-vars.sh
```

ParallelAccelerator running time compared to plain Julia and other implementations (sections 6.1-6.10 of the paper)

Running all versions of all the workloads in the paper would take a very long time, particularly if you don't have access to a 36-core machine, so instead we'll give a tour of a particular workload, the 2D wave equation simulation workload (described in sections 3.3 and 6.8 of the paper), known as *wave-2d* to its friends. Benchmarking the other workloads involves a process similar to that described below for *wave-2d*.

```
$ cd benchmarking/wave-2d/benchmarking/
$ ./benchmark.sh 1> totaltimes.log 2> offloadtimes.log
```

Each workload comes with a `benchmark.sh` script that will run various versions of the workload under various configurations of ParallelAccelerator and Julia. We also have MATLAB implementations of most workloads, and Python or C/C++ implementations of a few. For *wave-2d*, we have ParallelAccelerator, plain Julia, and MATLAB implementations.

The `benchmark.sh` script for *wave-2d* will run five versions of the workload:

- a ParallelAccelerator version run with 36 threads (`@acc (36 threads)`), the code for which is in
  - `~/.julia/v0.5/ParallelAccelerator/examples/wave-2d/wave-2d.jl`
- the same ParallelAccelerator version run with 1 thread (`@acc (1 thread)`), same code as above
- a plain Julia version (`Julia 0.5 (1 thread)`), the code for which is in
  - `~/.julia/v0.5/ParallelAccelerator/examples/wave-2d/wave-2d.jl`
- a MATLAB version using the default MATLAB settings (`Matlab`), the code for which is in
  - `benchmarking/wave-2d/src/wave.m`
- the same MATLAB version run using the `-singleCompThread` option (`Matlab (1 thread)`), same code as above

These five versions correspond to what's shown in the *wave-2d* subfigure of Figure 8 of the paper. Let's try running them!

(Note: if you have MATLAB, you can edit `env-vars.sh` to point to your MATLAB executable; if not, you should be able to run `benchmark.sh` anyway, and it will log failures for those runs. A free 30-day trial version of MATLAB is available.)

`benchmark.sh` will run each version five times; to change this, change the `NUM_TRIALS` environment variable at the top of `benchmark.sh`. In particular, if running the benchmarks is taking more time than you have, consider adjusting `NUM_TRIALS` to something smaller.

The `benchmark.sh` script will take some time to run and will produce a lot of output. If you run it with the above command (`./benchmark.sh 1> totaltimes.log 2> offloadtimes.log`), the log output will go two files, `totaltimes.log` and `offloadtimes.log`.

If all goes well, when `benchmark.sh` is done running, the contents of `totaltimes.log` will look something like this (as before, on an 8-core machine with 8 GB RAM):

```
Starting benchmark run
Package compilation time: 31.182489611
SELFPRIMED 6.365216996
SELFTIMED 0.916232509
Ending benchmark run
Starting benchmark run
Package compilation time: 31.058261159
```

```
SELFPRIMED 6.330945695
SELFTIMED 0.886127078
Ending benchmark run
Starting benchmark run
Package compilation time: 31.545143135
SELFPRIMED 6.227698681
SELFTIMED 0.904371725
Ending benchmark run
Starting benchmark run
Package compilation time: 31.124826062
SELFPRIMED 6.245557891
SELFTIMED 0.996313526
Ending benchmark run
Starting benchmark run
Package compilation time: 31.171427272
SELFPRIMED 6.241428397
SELFTIMED 0.98268331
Ending benchmark run
[...]
```

Notice that the benchmarking system measures package compilation time separately from `SELFPRIMED`. The time reported by `SELFPRIMED` is therefore only the time it takes ParallelAccelerator to compile the workload. (We accomplish this by compiling and running a dummy function through ParallelAccelerator before benchmarking.)

You can manually inspect the results in `totaltimes.log` if you like, but in the `benchmarking/util` directory, we provide a Python script called `munge-results.py` for processing these log files and generating benchmarking reports and figures.

The `munge-results.py` script assumes Python 2.7.x and depends on matplotlib (see directions for installing matplotlib on various platforms).

Once `benchmark.sh` finishes, you can run `munge-results.py` from the `benchmarking/wave-2d/benchmarking` directory as follows:

```
$ ../../util/munge-results.py totaltimes.log offloadtimes.log
```

Doing so will print a summary of benchmarking results to stdout:

```
Benchmarking report for wave-2d
================================================

@acc (36 threads)
Run 1
   running time:               0.916232509
   compile time (@acc):        6.365216996
   (no offloaded computation)
Run 2
   running time:               0.886127078
   compile time (@acc):        6.330945695
   (no offloaded computation)
Run 3
   running time:               0.904371725
   compile time (@acc):        6.227698681
```

```
   (no offloaded computation)
Run 4
   running time:               0.996313526
   compile time (@acc):        6.245557891
   (no offloaded computation)
Run 5
   running time:               0.98268331
   compile time (@acc):        6.241428397
   (no offloaded computation)

Average running time over 5 runs: 0.928937443
Average compile time over 5 runs: 6.26806742233
[...]
```

The "(no offloaded computation)" in the benchmarking report refers to the fact that all of the workload was run on CPU cores rather than being offloaded to an accelerator such as Xeon Phi. The average running time shown is the average over five runs with the first and last runs discarded.

Finally, if all goes well, the `munge-results.py` script will also produce a PDF figure in the file `/tmp/results/wave-2d/wave-2d-YYYY-MM-DD.pdf`, where `YYYY-MM-DD` is today's date. With any luck, this generated figure should resemble the *wave-2d* subfigure of Figure 8 of the paper – modulo differences in hardware and environment.

Impact of individual ParallelAccelerator optimizations (section 6.11 of the paper)

Finally, we consider how parallelism and individual compiler optimizations contribute to the speedup that ParallelAccelerator enables on each workload. Again, we take the *wave-2d* workload as an example. The process is similar for other workloads.

In the `benchmarking/util` directory, we provide a Julia script called `configtest.jl` for running ParallelAccelerator workloads in various configurations. Assuming that you've already run `source benchmarking/util/env-vars.sh`, you can run `configtest.jl` from the artifact directory as follows:

```
$ $JULIA_0_5_EXE benchmarking/util/configtest.jl --benchmarks-dir=benchmarking --workloads=wave-2d > b
```

When `configtest.jl` is done running, the contents of the `optimizations.log` file will look something like the following:

```
BENCHMARK SHORT NAME: wave-2d
Starting configuration
OMP_NUM_THREADS = 1
PIRSetFuseLimit = 0
PIRHoistAllocation = 0
vectorizationlevel = 1
Starting benchmark run
Package compilation time: 30.099966737
SELFPRIMED 8.612576799
SELFTIMED 0.573468455
Ending benchmark run
Starting benchmark run
Package compilation time: 29.85335769
SELFPRIMED 8.59620492
```

```
SELFTIMED 0.585841936
Ending benchmark run
Starting benchmark run
Package compilation time: 29.837718522
SELFPRIMED 8.719108591
SELFTIMED 0.615421742
Ending benchmark run
Starting benchmark run
Package compilation time: 30.521477947
SELFPRIMED 8.712099175
SELFTIMED 0.588877207
Ending benchmark run
Starting benchmark run
Package compilation time: 29.731774838
SELFPRIMED 8.554199595
SELFTIMED 0.584111302
Ending benchmark run
Ending configuration
[...]
```

If you've previously run the `benchmark.sh` script for *wave-2d* to generate the `totaltimes.log` and `offloadtimes.log` files, you can now use `munge-results.py` to process `optimizations.log` together with the previous two log files:

```
$ ../../util/munge-results.py totaltimes.log offloadtimes.log --opts-log=optimizations.log
```

This will produce a PDF figure in the file `/tmp/results/wave-2d/wave-2d-YYYY-MM-DD-opts.pdf`, where `YYYY-MM-DD` is today's date. This generated figure should resemble the *wave-2d* subfigure of Figure 9 of the paper, modulo differences in hardware and environment.