OOPSLA Artifact - Paper #59

Static Placement of Computation on Heterogeneous Devices

July 8, 2017

1 Introduction

This artifact contains the Etino tool described in the paper, along with scripts to run it and the various experiments we ran. Everything is packaged in a Docker container. We used Docker instead of a Virtual Machine because `nvidia-docker` allows us to easily access the GPU in the host machine, which we need.

In order to use the artifact out-of-the-box, you will need a 64-bit machine running Linux and have a NVidia Tesla GPU. The `pgcc` compiler, which we use to generate GPU-targeting code, supposedly supports other GPUs as well, even Radeon ones. However, since all GPUs we have readily available are Tesla ones we could not test it on other setups.

2 Set up

First, the host machine must have two important pieces of software: the latest CUDA driver and `nvidia-docker`, which we'll use to run our container with access to your NVidia Tesla GPU.

The CUDA driver can be downloaded here: `https://developer.nvidia.com/cuda-downloads`. Several Linux distributions are officially supported. If yours is not, it is very likely that the maintainers of your distro package the driver in some other source.

`nvidia-docker` can be downloaded by following the instructions from their Github page: `https://github.com/NVIDIA/nvidia-docker`. Again, instructions are distribution-specific. However, this time there is one option that should work for all distributions.

After you have downloaded and installed both the CUDA driver and `nvidia-docker`, we can already run our Docker image with Etino. Thus, run:

```
gabriel@britto:~/$ sudo nvidia-docker run -it gpoesia/etino
```

This will download the image from the Docker registry (which will take a while since the image is quite large, around 10GB) and start a container running that image. The container will be started in interactive mode (-it flags), and thus we should get a root shell inside the container:

```
gabriel@britto:~/$ sudo nvidia-docker run -it gpoesia/etino
[sudo] password for gabriel:
root@3d9b5f45b9af:/artifact#
```

Let's first check that the NVidia GPU is accessible from within Docker. For that, run `nvidia-smi`:

```
root@3d9b5f45b9af:/artifact# nvidia-smi
Fri Jul  7 01:25:42 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.26                 Driver Version: 375.26
              |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile
    Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util
    Compute M. |
|===============================+======================+======================|
|   0  Tesla C2070         Off  | 0000:01:00.0     Off |
             0 |
| 30%   57C   P12    N/A /  N/A |      0MiB /  5296MiB |      0%
       Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:
    GPU Memory |
|  GPU       PID  Type  Process name
    Usage      |
|=============================================================================|
|  No running processes found
              |
+-----------------------------------------------------------------------------+
```

Here, we can see we have the NVidia driver version 375.26, and a Tesla C2070 GPU. You should see your GPU's model here. Otherwise, please check that CUDA in the host machine is working properly. Also, no running process should be using the GPU. Otherwise, that might interfere with the resuls.

Now that we have checked that the GPU is up and running, let's try to run Etino. Etino was implemented on top of the existing dawncc source-to-source compiler. Its root directory is "/artifact/etino/src/OptimalCallGraph" directory. Let's cd there and have a look at the files:

```
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph# ls
3fold           CMakeLists.txt  benchmarks                core
out_da.log      parallelize.sh  run_cross_validation.sh
    run_on_benchmark.sh
src             3fold_fold1     CodeTransformer
    compare_results.py
experiment.log  out_div.log     results_britto.tar.gz
    run_experiment.sh
sa_iterations   tools           fold_fold2                README.md
comparison_dawncc_britto_cc.log graph.dot                 output
results_eos     run_on_all_benchmarks.log
    sa_iterations_fold1
3fold_fold3     a.ot            context_examples          log
parallel_loop_anotations        results_seq
    run_on_all_benchmarks.sh
simulated_annealing.log
```

There are many files here. We won't need to directly use most of them, however.

For now, let's make sure the essential pieces are working by testing Etino on a single benchmark. In the next section, we'll see how to reproduce the experiments more thoroughly.

The original benchmarks lie in the "benchmarks" directory. This directory has one sub-directory for each of the three benchmark suites we used:

```
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph# ls
    benchmarks/
BenchmarkGame  DataMining  MgBench  Polybench  README.txt
    default_makefile  run_all.sh
```

Let's try one of the original benchmarks in there: Polybench/2MM. Each benchmark has its own directory with its source code files, and a Makefile to build it. All Makefiles produce an executable called "run_benchmark". To compile and run 2MM, we first go to its directory, run "make" (some warnings might show up, but they can be safely ignored) and then run the generated program:

```
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph/benchmarks/Polybench/2MM#
    make
rm -f run_benchmark *.o *.oo
pgc++ ./2mm.c  -lm -fast -Mipa=fast,inline -Msmartalloc
    -DINPUT_SIZE= -o run_benchmark
"./2mm.c", line 74: warning: incompatible redefinition of macro
    "INPUT_SIZE"
  #define INPUT_SIZE 3000
          ^

"./2mm.c", line 205: warning: variable "t_start_GPU" was declared
    but never
          referenced
    double t_start, t_end, t_start_GPU, t_end_GPU;
                           ^

"./2mm.c", line 205: warning: variable "t_end_GPU" was declared
    but never
          referenced
    double t_start, t_end, t_start_GPU, t_end_GPU;
                                        ^

IPA: Recompiling 2mm.o: new IPA information
run_and_time run_benchmark ./2mm.c  --compile
259687852320866934458691341577869423220 =>
    310997910275016979522321399301534439322
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph/benchmarks/Polybench/2MM#
    time ./run_benchmark
<< Linear Algebra: 2 Matrix Multiplications (D=A.B; E=C.D) >>
CPU Runtime: 418.229967s

real    6m58.368s
user    6m58.248s
sys     0m0.188s
```

If you get similar output, this confirms the compiler we need is properly licensed. The image ships with the “pgcc” compiler and a Community Edition license supposed to work until June 2018. If you are using the image after that date, you may need to license pgcc again. Please head to PGI’s Web site for further instructions: `http://www.pgroup.com/`

Now, let’s try to run a GPU-accelerated version of this same benchmark. First, let’s go back to Etino’s root directory:

```
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph/benchmarks/Polybench/2MM#
    cd ../../../
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph#
```

When we run Etino (which we will do soon), it will write optimized programs to the “output” directory, which has a similar structure to the “benchmarks” directory. For now, let’s compile and run the optimized version of 2MM which is already there:

```
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph# cd
    output/benchmarks/Polybench/2MM/
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph/output/benchmarks/Polybench/2MM#
    make
rm -f run_benchmark *.o *.oo
pgc++ ./2mm.c  -lm -fast -Mipa=fast,inline -Msmartalloc -acc
    -ta=nvidia:cc20 -o run_benchmark
"./2mm.c", line 298: warning: variable "t_start_GPU" was declared
    but never
          referenced
    double t_start, t_end, t_start_GPU, t_end_GPU;
                           ^

"./2mm.c", line 298: warning: variable "t_end_GPU" was declared
    but never
          referenced
    double t_start, t_end, t_start_GPU, t_end_GPU;
                                        ^

"./2mm.c", line 332: warning: variable "t_start_GPU" was declared
    but never
          referenced
    double t_start, t_end, t_start_GPU, t_end_GPU;
                           ^

"./2mm.c", line 332: warning: variable "t_end_GPU" was declared
    but never
          referenced
    double t_start, t_end, t_start_GPU, t_end_GPU;
                                        ^

IPA: Recompiling 2mm.o: new IPA information
run_and_time ./run_benchmark ./2mm.c  --compile
181935164727272483943793794146591661330 =>
    104987213579557457581296620630419916324
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph/output/benchmarks/Polybench/2MM#
    time ./run_benchmark
<< Linear Algebra: 2 Matrix Multiplications (D=A.B; E=C.D) >>
CPU Runtime: 0.000999s

real    0m1.194s
user    0m0.724s
sys     0m0.348s
```

It all works as expected, this should take much less than the previous run. One possible error you might encounter here is that pgcc complain that the code was compiled to an accelerator which is not available. Such error will look like this:

```
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph/output/benchmarks/Polybench/2MM#
    ./run_benchmark
<< Linear Algebra: 2 Matrix Multiplications (D=A.B; E=C.D) >>
Current file:
    /artifact/etino/src/OptimalCallGraph/output/benchmarks/Polybench/2MM/./2mm.c
        function: _Z18GPU__main__mm2_cpuPfS_S_S_S_
        line:     202
Current region was compiled for:
  NVIDIA Tesla GPU sm20
Available accelerators:
  device[1]: Native X86
The accelerator does not match the profile for which this program
    was compiled
```

This is because we need to specify the CUDA Capability of the GPU we're running on when we compile with pgcc. Our GPUs have CUDA Capability 2.0, and our Makefile informs that to pgcc. In order to change this, we need only to edit the "default_makefile" file in the "output/benchmarks" directory. Its second line is the one you need to change:

```
root@3d9b5f45b9af:/artifact/etino/src/OptimalCallGraph/output/benchmarks/Polybench/2MM#
    cat ../../default_makefile
OPTS=-lm -fast -Mipa=fast,inline -Msmartalloc
OPENACC=-acc -ta=nvidia:cc20
COMPILER=pgc++
...
```

Simply change "cc20", to cc30, cc40, cc50 or cc60 if your CUDA Capability is 3.0, 4.0, 5.0 or 6.0, respectively.

We managed to run a program accelerated using the GPU. But the program was already there. Let's now run Etino and produce the program again.

In the paper, we describe the procedure we use for calibrating a cost model for Etino. Here, however, we'll use a pre-learned set of parameters. We'll see how to run Simulated Annealing in the "Step-by-step" section.

The script that fires the entire Etino pipeline is "run_benchmark.sh". It will first run the analyses implemented as LLVM passes in the 'src' directory, then use the produced information to run two clang tools to create source-level function clones (those being implemented in the "CodeTransformer" directory), then call the dawncc compiler to generate OpenACC directives which pgcc uses to offload computation to the GPU. To run the script, we simply need to pass it the directory in which the desired benchmark lies. It will print a number of debugging messages, which you can ignore. In the end, it should print some statistics about the program, including the number of loops that were annotated with OpenACC directives in the output source code.

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# bash
    run_on_benchmark.sh benchmarks/Polybench/2MM

[...] (Huge output)

Found parallel loop in file 2mm.c, line 224
Found parallel loop in file 2mm.c, line 238
Found parallel loop in file 2mm.c, line 247

Writing output to file
    /artifact/etino/src/OptimalCallGraph/output/benchmarks/Polybench/2MM/2mm_AI.c
===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

200 PTRRangeAnalysis - Number of memory access
200 PTRRangeAnalysis - Number of memory analyzed access
  2 lcssa            - Number of live out of a loop variables
 50 loop-rotate      - Number of loops rotated
 55 mem2reg          - Number of PHI nodes inserted
 84 mem2reg          - Number of alloca's promoted
 56 mem2reg          - Number of alloca's promoted with a single
    store
  4 mem2reg          - Number of dead alloca's removed
136 region           - The # of regions
100 region           - The # of simple regions
 50 scalar-evolution - Number of loops with predictable loop counts
 50 writeExpressions - Number of analyzable loops
 50 writeExpressions - Number of annotated loops
 50 writeExpressions - Number of loops
```

Now we can go back to the output/benchmarks/Polybench/2MM directory, where the output was written, and test the benchmark again.

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph/output/benchmarks/Polybench/2MM#
    make
rm -f run_benchmark *.o *.oo
pgc++ ./2mm.c  -lm -fast -Mipa=fast,inline -Msmartalloc -acc
    -ta=nvidia:cc20 -o run_benchmark
[...] (Warnings omitted)
IPA: Recompiling 2mm.o: new IPA information
run_and_time ./run_benchmark ./2mm.c  --compile
109271919204393624325889752849542064067 =>
    284373769166792146446953318402306369758
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph/output/benchmarks/Polybench/2MM#
    time ./run_benchmark
<< Linear Algebra: 2 Matrix Multiplications (D=A.B; E=C.D) >>
CPU Runtime: 0.000987s

real    0m1.186s
user    0m0.719s
sys     0m0.341s
```

As a last test, let's run the same program using one important tool for Simulated Annealing to run: `run_and_time`. This is a tool that will run the executable 5 times, average the running times, and save that to a local database, associating this data with a hash of the executable, command line options and input files. This result is cached for when we run an identical program for a second time with the same inputs. Since this happens a lot during the training phase, this tool is very important for we to get results timely. This tool is integrated in our Makefiles. To run it, just `make run`:

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph/output/benchmarks/Polybench/2MM#
    make run
run_and_time ./run_benchmark
1.191
```

It will take more than the other run since it will run the program 5 times. After that, a second run should return near instantly.

We are done! If you managed to run everything up to here, it means you have a working setup of Etino, pgcc and the benchmark suites we have used. In the next section, we'll se how to reproduce the paper experiments.

3 Step-by-step: reproducing our experiments

As described in Section 5 of the paper, the first step is to run Etino's callibration procedure. In the experiments, we have ran 3-fold cross-validation. This process is all orchestrated by the `run_cross_validation.sh` script. Just run it without parameters from Etino's root directory and it will begin:

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# bash
    run_cross_validation.sh
Iteration time: 1h34m19s
Iteration 0 train_speedup -24.87964655172019 test_speedup
    -8.167188073360562 solution:  -param-parallelism=9.668
    -param-kernel-creation-cost=2358.093
    -param-array-transfer-cost=5815.250
    -param-loop-iterations=3.040 -param-branch-cost=9.699
Iteration time: 25m53s
Iteration 1 train_speedup -24.879400306952892 test_speedup
    -8.167188073360562 solution:  -param-parallelism=10.027
    -param-kernel-creation-cost=3283.728
    -param-array-transfer-cost=6324.522
    -param-loop-iterations=2.267 -param-branch-cost=6.944
Iteration time: 2m16s
Iteration 2 train_speedup -24.879400306952892 test_speedup
    -8.167188073360562 solution:  -param-parallelism=7.299
    -param-kernel-creation-cost=3945.793
    -param-array-transfer-cost=8490.937
    -param-loop-iterations=2.837 -param-branch-cost=4.842
Iteration time: 25m38s
Iteration 3 train_speedup -24.87868395853893 test_speedup
    -8.167188073360562 solution:  -param-parallelism=5.000
    -param-kernel-creation-cost=3602.848
    -param-array-transfer-cost=5998.017
    -param-loop-iterations=3.072 -param-branch-cost=3.836
Iteration time: 34m27s
Iteration 4 train_speedup -24.79055770514988 test_speedup
    -8.034345786595203 solution:  -param-parallelism=6.023
    -param-kernel-creation-cost=2525.526
    -param-array-transfer-cost=6667.370
    -param-loop-iterations=2.182 -param-branch-cost=5.220
Iteration time: 2m13s
Iteration 5 train_speedup -24.79055770514988 test_speedup
    -8.034345786595203 solution:  -param-parallelism=8.605
    -param-kernel-creation-cost=2716.393
    -param-array-transfer-cost=8942.093
    -param-loop-iterations=2.625 -param-branch-cost=4.763
[...]
```

A word of caution: full 3-fold Simulated Annealing takes a long time to run in our setup (about 6 hours for each fold, totalling about 18 hours overall). The first iterations take specially long (typically around 30 minutes each), since the cache – mentioned in the Set-up section – will be fresh, and in each step all benchmarks are compiled and ran 5 times (there are 30 benchmarks, and some of them run for a couple of minutes in their original sequential version). After the first 10 or so iterations, each next iteration usually takes about 2 minutes to run (which is essentially the time to compile all benchmarks and hit the cache when running them).

We ran 100 iterations on each fold. If you only want to see Simulated Annealing running to completion without waiting that long, you may lower that number of iterations to e.g. 10. For that, you need to edit the "tools/optimize_cost_model.py" file. The last line contains the number of iterations, which you can change to whatever you want. Bear in mind, however, that our results were obtained with 100 iterations. Less than that might not be enough for the calibration to converge to a good cost model for your architecture.

Another trick you can use to get results faster is to ignore some benchmarks. The artifact ships with a premade 3-fold split stored in the `3fold_foldX` files (for $X = 1, 2, 3$). You can open these files and leave just 2 benchmarks in each fold, for instance. Each benchmark must be in one line, and you only need its name (not its full path or the suite's name). For example:

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph#
    cat 3fold_fold1
2DCONV
NsieveBits
COVAR
SYR2K
SYRK_M
2MM
GEMM
ATAX
cholesky
k-nearest
mat-sum
str-matching
```

After Simulated Annealing finishes running, you can then see which were the final values of the parameters in each fold by looking at the logs in 3fold/ directory. There will be one subdirectory for each fold. Each of these subdirectories will have one file for each iteration, and one `simulated_annealing.log` file, containing the parameters and the values of the Quality in the training and test sets. These values were used to plot Figure 9. For each iteration, you will find a file named `iteration_X.log` which has a comparison between the speed-ups dawncc and Etino gain over the sequential program for all benchmarks, divided in training and test sets. The test set results in the last iterations were used to plot Figure 10.

For example, these are the first lines of `simulated_annealing.log` for one of the folds:

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# head -n3
    3fold/sa_iterations_fold2/simulated_annealing.log
Iteration 0 train_speedup 0.05864048500456033 test_speedup 0.0
    solution:  -param-parallelism=9.668
    -param-kernel-creation-cost=2358.093
    -param-array-transfer-cost=5815.250
    -param-loop-iterations=3.040 -param-branch-cost=9.699
Iteration 1 train_speedup 0.05864048500456033 test_speedup 0.0
    solution:  -param-parallelism=10.027
    -param-kernel-creation-cost=3283.728
    -param-array-transfer-cost=6324.522
    -param-loop-iterations=2.267 -param-branch-cost=6.944
Iteration 2 train_speedup 0.05864048500456033 test_speedup 0.0
    solution:  -param-parallelism=7.299
    -param-kernel-creation-cost=3945.793
    -param-array-transfer-cost=8490.937
    -param-loop-iterations=2.837 -param-branch-cost=4.842
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# tail -n3
    3fold/sa_iterations_fold1/simulated_annealing.log
Iteration 98 train_speedup 2.6928391974681385 test_speedup
    3.0508173100868934 solution:  -param-parallelism=14.152
    -param-kernel-creation-cost=3306.381
    -param-array-transfer-cost=686.947
    -param-loop-iterations=13.063 -param-branch-cost=12.326
Iteration 99 train_speedup 2.6928391974681385 test_speedup
    3.0508173100868934 solution:  -param-parallelism=17.274
    -param-kernel-creation-cost=3430.706
    -param-array-transfer-cost=930.569
    -param-loop-iterations=13.137 -param-branch-cost=8.430
Final solution:  train_speedup 2.6928391974681385 test_speedup
    3.0508173100868934 solution:  -param-parallelism=16.309
    -param-kernel-creation-cost=3136.800
    -param-array-transfer-cost=652.575
    -param-loop-iterations=13.729 -param-branch-cost=6.429
```

And these are the results in the last iteration (with most benchmarks omitted for brevity):

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# cat
    3fold/sa_iterations_fold2/iteration_99.log
Comparing Eos against baseline Sequential
Dataset:  Test
./MgBench/cholesky: LOSS (2.53x)
./Polybench/SYRK_M: TIE
./MgBench/mat-sum: TIE
./Polybench/GEMM: WIN (11.90x)
[...]
Test Wins:  3
Test Losses:  1
Test Ties:  7
Test Failures:  0
Test W/L Ratio:  3.0
Test Average speed-up:  3.0508173100868934
Dataset:  Train
./Polybench/GRAMSCHM: TIE
./MgBench/collinear-list: LOSS (1.18x)
./Polybench/3MM: WIN (33.22x)
[...]
Train Wins:  3
Train Losses:  2
Train Ties:  14
Train Failures:  0
Train W/L Ratio:  1.5
Train Average speed-up:  2.6928391974681385
```

The reported "Average speed-up" is, in fact, the Quality as defined in Section 4 of the paper.

The `run_on_benchmark.sh` script allows us to run Etino with different parameters for its cost models. This is done via the `OPTIMALCALLGRAPH_OPTS` environment variable. The Etino LLVM passes take the parameters as command-line arguments. The parameters we need to pass are exactly those output in the `simulated_annealing.log` file. For example, if we want to run Etino with the parameters from the 99-th iteration of fold 2 in the run above, we only need to set:

```
# export OPTIMALCALLGRAPH_OPTS="-param-parallelism=16.309
    -param-kernel-creation-cost=3136.800
    -param-array-transfer-cost=652.575
    -param-loop-iterations=13.729 -param-branch-cost=6.429"
# bash run_on_benchmark.sh <<benchmark>>
```

If these parameters are not passed, Etino runs with a default set of values for the parameters, taken from one of our simulated annealing runs.

We also have the `run_on_all_benchmarks.sh` script which simply calls `run_on_benchmark.sh` for all benchmarks.

When processing a benchmark, Etino produces an intermediary file with its decisions, which can also be used to calculate statistics about the program. These files are also used to use Etino as a guiding tool, as in Sections 5.3 and 5.5. This is the file for our running example, Polybench/2MM:

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# cat
    output/benchmarks/Polybench/2MM/transformations
CLONE init_array CPU__main__init_array
CLONE init_array GPU__main__init_array
CLONE rtclock CPU__main__rtclock
CLONE rtclock GPU__main__rtclock
CLONE mm2_cpu CPU__main__mm2_cpu
CLONE mm2_cpu GPU__main__mm2_cpu
CALL main init_array CPU__main__init_array
CALL main rtclock CPU__main__rtclock
CALL main mm2_cpu GPU__main__mm2_cpu
CALL main rtclock CPU__main__rtclock
CLONE main GPU__main
CALL GPU__main init_array CPU__main__init_array
CALL GPU__main rtclock CPU__main__rtclock
CALL GPU__main mm2_cpu GPU__main__mm2_cpu
CALL GPU__main rtclock CPU__main__rtclock
```

A "CLONE" instruction says that a certain function should be cloned. In the current implementation, every function is cloned once for each processor (if the clone ends up not being called, it will be stripped from the final binary). A "CALL A B C" instruction says that, in function A, all calls to B should be replaced by calls to C. This file is produced even if Etino is not able to automatically generate parallel code for the benchmark, due to dawncc not being able to infer array bounds to copy the data, for example. Thus, they may be used in a manual setting, as we did in Sections 5.3 and 5.5 with ppgc and pgcc as the compilers, respectively.

These "transformations" files are also used to generate Table 8. We have a script that processes those files and generates the LaTeX table:

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# python
    tools/generate_benchmark_statistics_table.py output/benchmarks/
\\begin{tabular}{|c|c|c|c|\}
\\hline \\
Benchmark & Orig. & CPU & GPU \\ \hline
output/benchmarks/DataMining/knn & 4 & 4 & 0 \\ \hline
output/benchmarks/DataMining/itemsets & 4 & 4 & 0 \\ \hline
output/benchmarks/DataMining/kmeans & 4 & 4 & 0 \\ \hline
output/benchmarks/BenchmarkGame/NBody & 4 & 4 & 0 \\ \hline
output/benchmarks/BenchmarkGame/PartialSums & 4 & 4 & 0 \\ \hline
output/benchmarks/BenchmarkGame/SpectralNorm & 4 & 4 & 0 \\ \hline
output/benchmarks/BenchmarkGame/NsieveBits & 4 & 4 & 0 \\ \hline
[...]
```

Note that the DataMining statistics are not correct, since these were not automatically processed (they do not figure in the output/ directory). Rather, the transformation files that were generated for these benchmarks lie in their original directories. Thus, to see the statistics corresponding to them, run the same script but pass “benchmarks/” instead of “output/benchmarks”. The transformation files that ship with DataMining were generated with full cloning (i.e. one clone for each calling context, not for each processor). The results, however, were equivalent to those in the paper - we did not see any improvements.

The experiments in Section 5.4 were generated using the `tools/max_slowdown.py`. To run it, simply pass the list of benchmarks to be tested as command-line arguments:

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# python
    tools/max_slowdown.py benchmarks/Polybench/*
('benchmarks/Polybench/2MM/', 1000, 2.5954201221466064,
    2.5717809200286865)
('benchmarks/Polybench/2MM/', 2000, 22.482841968536377,
    2.571237087249756)
('benchmarks/Polybench/2MM/', 3000, 92.98363089561462,
    2.586669921875)
[...]
```

The data is formatted as follows: each line has a tuple (benchmark, input size, sequential version running time, Etino’s version running time). This data was used to generate Figure 12 (right).

In this last experiment in particular, you can set the DAWNCC environment variable to 1 in order to obtain the same numbers for dawncc:

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# export
    DAWNCC=1
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# python
    tools/max_slowdown.py benchmarks/Polybench/*
[...]
```

Set it to 0 whenever you wish to be back on using Etino.

4 Adding benchmarks

In the last section, we ran over all the experiments we presented in the paper. This section is here if you want to apply Etino on other benchmarks.

To add a benchmark suite, simply create a new directory under "benchmarks". It will be easier if your benchmark can be compiled with the Makefile that compiles the existing programs (`benchmarks/default_makefile`). If so, just copy the Makefile from another benchmark. Otherwise, custom modifications will be needed depending on how your benchmark is layed out in the file system and what compilation flags you need in order to build it. Name your Makefile exactly `Makefile`, since the scripts automatically detect benchmarks by looking for directories under `benchmarks/` which have a file with this name. After you have properly created the new folder, the scripts you have used before should work out of the box.

One detail in adding benchmarks is in regarding the detection of parallel loops. dawncc comes with its own parallel loop analysis, and we can either rely on it or manually annotate parallel loops. Since their analysis is not 100% precise and our paper does not try to solve the problem of detecting parallel loops, we have reported results using manual annotations. These annotations lie in the `parallel_loop_anotations/parallel_loops` directory:

```
root@4c65230e7d42:/artifact/etino/src/OptimalCallGraph# head -n5
    parallel_loop_anotations/parallel_loops
2mm.c init_array 0,2,4,6
2mm.c mm2_cpu 0,3

2DConvolution.c conv2D 0
[...]
```

The format of this file is simple. Each line contains a triple: file name, function name, and which loops in that function are parallel. Loops are indexed by the order they appear in the source code, and indexes start in 0. In C and C++, **for**, **while** and **do..while** are all considered loops.

If you wish to try using Etino coupled with dawncc's parallel loop analysis, edit the `parallelize.sh` script, and look for the last **opt** invocation. Change the parameter `-Emit-Parallel=false` to `-Emit-Parallel=true`. That should do it.

5 Changing the code and/or the cost models

Etino's core algorithm is implemented as an LLVM analysis pass called `OptimalCallGraph`. The algorithm is implemented in `src/ComputationPlacementSolver.h/cpp`. It is parameterized by the CPU and GPU cost models which can be found under `src/CostModel.h/cpp`. You can change those and try to improve the overall results. After doing any changes, simply go to the `/artifact/etino/build` directory and type "make" to compile them. When you go back to `/artifact/etino/src/OptimalCallGraph`, all scripts will already use the new compiled implementation.