Enabling Julia to target GPGPUs using Polly

Student Information

Name: Singapuram Sanjay Srivallabh
E-Mail: singapuram.sanjay@gmail.com
University: Birla Institute of Technology and Science - Pilani, Hyderabad Campus, India
Major: Bachelor of Computer Science and Engineering.

Introduction

Polly

Polly is an analysis and optimization framework[1] for LLVM that uses polyhedral compilation techniques to optimize programs. It extracts information from LLVM-IR produced by language front-ends and analyses them for possible parallelism and data-locality optimizations using the polyhedral model. It can then generate parallelized CPU code or even GPU code, thus automating the offloading process. This improves developer productivity and helps target upcoming architectures without requiring developers to learn to program them. Polly is able to speed up compute kernels significantly[1], especially in the context of dense linear algebra and iterative stencil computations.

Polly-ACC

Polly-ACC[2] is an alternative datapath for LLVM-IR within Polly, for optimizing IR meant for GPUs. It interfaces with ppcg, a standalone transpiler that converts C/C++ code into CUDA or OpenCL code, and uses its optimization strategies for GPU code. The CPU specific passes like IslScheduleOptimizerPass and CodeGenerationPass are dropped in favour of the PPCGCodeGenerationPass.

The `@polly` macro

Polly[2] currently interfaces with Julia by optimizing Julia functions annotated with the `@polly` macro. This was a part of the work done by Matthias Reisinger in his GSoC 2016 project titled “Enabling Polyhedral Optimizations in Julia”[5] [6]

---

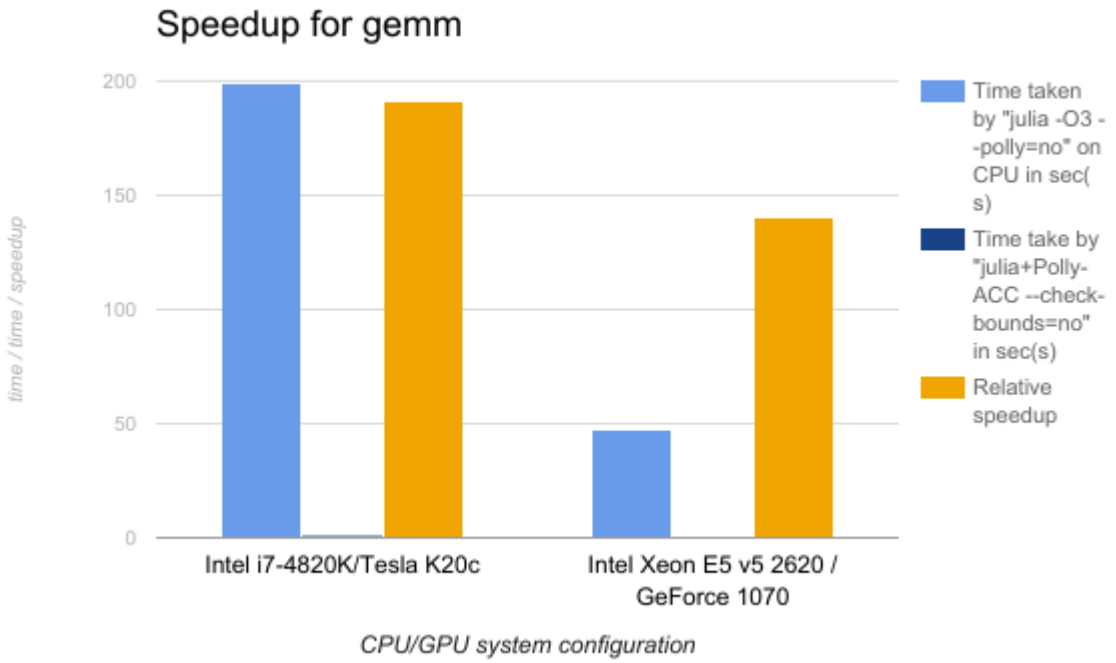
[1] Polly is a sequence of individual passes, each which is meant to analyse or transform the IR or inform the user about the IR.
[2] “Polly” might be interchangeably used with Polly-ACC in a context not discussing GPU-specific topics.

Julia + Polly-ACC: Proof of Concept and Motivating Benchmark

I have made initial progress in integrating Polly-ACC into Julia, which can be found here (as partial diffs and benchmark results). These changes include code to handle LLVM-IR generated by a debug build of Julia and also ensures certain runtime libraries are linked.

Benchmarks

The Julia gemm kernel defined in PolyBench.jl was benchmarked with and without Polly-ACC enabled, and Julia specific optimizations enabled when Polly-ACC wasn't. Matrices of sizes 2300 and 11448 were used to compare the speeds of CPU-only and GPU-only implementations. The benchmark revealed speedups of at least 141x upto 191x. More information can be found in this Sheet associated with the graph or at the end of this document.



Benefits to the Community

As is evident from the preliminary benchmark results, offloading kernels to GPUs can have an incredibly positive effect on the runtime, especially for programmers or researchers who don't want to be burdened by learning yet another programming language. Given that many people are getting into computational finance and data science and a lot of personal systems now integrate GPUs, this research can now be accelerated, and is accelerated further in the future when hardware gets faster and software gets smarter.

Deliverables

This section outlines the core objectives, *Must haves*, and optional outcomes, *Nice to haves*, of this project. I'll be tackling the *Nice to haves* in case I have time and 'am done with the *Must haves*.

Must haves

- Enable polly-acc for all kinds of builds
- Use `@polly-acc` to compile code for the GPU and `@polly` for the CPU
- Enable Julia to send runtime information to Polly-ACC to generate the most suitable code for the set of run-time parameters.

Nice to haves

- Make Polly default to optimizing for CPU in case GPU cost model suggests a costlier offload.
- Fix OpenMP code generation of Polly for Julia.

Approach and Implementation Plan

1. Adapting Julia and Polly to include components and dependences of Polly-ACC

The Julia and Polly codebases have to be modified such that components and dependencies of Polly-ACC are always included in the build and it functions properly in any scenario.

Polly inserts calls to runtime functions that handle device-host data transfers and launch kernels. These are a part of libGPURuntime.so, found in Polly, which has to be linked to libjulia.so or to libLLVM.so.

Polly-ACC fails to create a GPU kernel whenever the associated LLVM module has metadata in the form of debug-intrinsic instructions. The local Polly codebase currently manages it with debug-info stripping functions. I have to find a way that concretely ensures that debug-intrinsics aren't included in functions meant for Polly, because even OpenMP code generation might fail when such instructions are present. This could entail changes in Julia codebase itself, which would prevent debug intrinsics in functions meant for Polly.

Currently, Julia initializes just the LLVM back-end that targets the native architecture. Since Polly-ACC uses the NVPTX-backend (or AMDGPU backend) to produce kernel code, even this backend must be initialized whenever Julia needs to use Polly-ACC. This can be done inside either Julia or Polly.

2. Introduction of `@polly-acc` macro

This macro would help indicate functions that are suitable for GPUs. The behaviour would be similar to that of `@polly` macro when Julia's passed "--polly-target=gpu" via JULIA_LLVM_ARGS. Instead of considering all `@polly` annotated functions for GPU offload, GPU suitable functions can be separately considered by `@polly-acc`. `@polly` can then be reserved for functions that'd do better when optimized for CPUs.

This'd require Polly to register 2 different sequence of passes with Julia, one for `@polly` and another for `@polly-acc`.

I could have `@polly-acc` default to `@polly`'s functionality in case the cost model indicates that the function isn't well suited for a GPU offload. This can be done by splitting Polly's monolithic GPUCodegenerationPass into an analysis and a transformation pass, and having the analysis pass right after Polly's ScopInfo[3] pass and just before all other transformation passes.

3. Getting more Julia generated kernels recognized and optimized

For the `@polly-acc` macro to be widely applicable, it is important that it is able to recognize and optimize certain widely used compute kernels which are representative of algorithms used in compute intensive programs. PolyBench.jl is such a set of kernels for polyhedral optimizers, especially Polly, written in Julia. I would be working to make Polly recognize and optimize a subset of these algorithms.

This step would be analyzing the reasons for certain kernels not being optimized by Polly-ACC. Expanding the Polly's envelope to these kernels could entail changes to (in decreasing order of likelihood),

1. Julia LLVM-IR code generation
2. Polly-ACC's GPU cost model
3. Polly's SCoP[4] detection algorithm
4. ppcg's optimization strategy

The best possible way to capture these changes (not including those for ppcg) would be a Julia module for Polly. This way, the Julia and Polly codebases can avoid tighter coupling and remain universal for as many modules or front-ends respectively. I would be building on the polly.jl module proposed by Matthias Reisinger's pull request.

---

[3] Polly's ScopInfo pass - stores a part of program in representation suitable for polyhedral analysis

[4] SCoP - Static Control Part - a portion of the program that can be analyzed and optimized by polyhedral techniques. E.g. for loops with finite number of iterations and constant iterator update: for(i=0;i<N;i+=C){...}. Such a SCoP has atleast 2 paramaters, in the form of the loop bound N and update value C.

4. Using run-time parameters to better optimize generated code

This step envisages Polly to be able generate code that best fits a particular class of problem sizes. For e.g. it might be better off running gemm as OpenMP code, when the matrices are small, than offloading it to the GPU which would require to store both OpenMP code and the NVPTX kernel. We could even have different NVPTX implementations of the same SCoP which ppcg can generate based the information it has about variables in the code (which include loop bounds). As a start we could consider those SCoPs which have small number of parameters

There are 2 challenges to this step,

1. I have to find a cost effective way to associate a function's runtime parameters to a particular machine code definition, which is most suitable for those parameters
    - Association strategy must incorporate ppcg's behaviour in order to correctly map the parameters to existing function definition
2. Julia stores only one machine code definition per function. To overcome this challenge,
    - I could modify Julia's source to store multiple machine code definitions of the same function, which has been annotated with `@polly` or `@polly-acc`
    - Or circumvent the problem by appending machine code to and updating existing definition of the function and include the association strategy as a part of the function.

This would be the most complex and time-consuming step in the project.

Timeline

This is a preliminary plan. It would undergo changes with further interaction with the community and my mentor and as my understanding of the goals evolve. Since this aligns exactly with my summer vacations, I'll be available on all days throughout the week.

Time Span	Activity
April 4 - May 29	*Community Bonding Period* Delving deeper into the Polly, Julia and ppcg codebases ; intensify contact with the community
Coding officially begins!	
May 30 - Jun 6	Adapting Julia and Polly to include components and dependencies of Polly-ACC; Testing Polly-ACC in various build scenarios
Jun 7 - Jun 13	Introduce the `@polly-acc` macro; direct pass manager to schedule different passes for `@polly` and `@polly-acc`
Jun 14 - Jun 26	Define the set of benchmarks that are expected to be

	optimized by Polly; Investigate reasons behind failure of certain benchmarks; Propose changes to appropriate repositories
Jun 27 - Jun 30	Continue work; Prepare and submit Phase 1 report
Phase 2 starts	*Phase 1 evaluation*
July 1 - July 7	[buffer] Investigate reasons behind failure of certain benchmarks; Propose changes to appropriate repositories
July 8 - July 14	Analyze `ppcg` source and come up with cost effective parameters-to-code mapping strategy; consider cost-models from Polly-ACC and Polly's CPU-optimization passes
July 15 - July 20	Evaluate implementation options of *using run-time parameters to better optimize generated code* with communities
July 21 - July 28	Continue working; Prepare and submit Phase 2 report
Final Phase starts	*Phase 2 evaluation*
July 29 - Aug 4	Implementation of *using run-time parameters to better optimize generated code*
Aug 5 - Aug 11	Testing implementation for quality ; benchmarking specific set of kernels on different problem sizes.
Aug 12 - Aug 18	Continue benchmarking ; Implement *nice to have* features ; Start documentation
Aug 19 - Aug 25	Implement *nice to have* features ; continue documentation ; code cleanup ; start preparing for final evaluation
Aug 26 - Aug 29	Prepare and submit final evaluation

I've already started working on some aspects of the project and believe that I would be well equipped with required knowledge when GSoC starts. I have left a week long buffer (July 1st - 7th) in case I attend JuliaCon (June 20th - 24th) to deliver a talk on Julia and Polly, during which I can work in the evenings.

Future Directions

Polly-ACC can be extended to generate code that can run on many GPUs in a single node. This could be done by partitioning the iterations (or iteration space) across GPUs. A cost model should be in effect to allow such partitioning only when iteration space is too large for a single GPU, and there can be benefit from being run on several GPUs with negligible overheads.

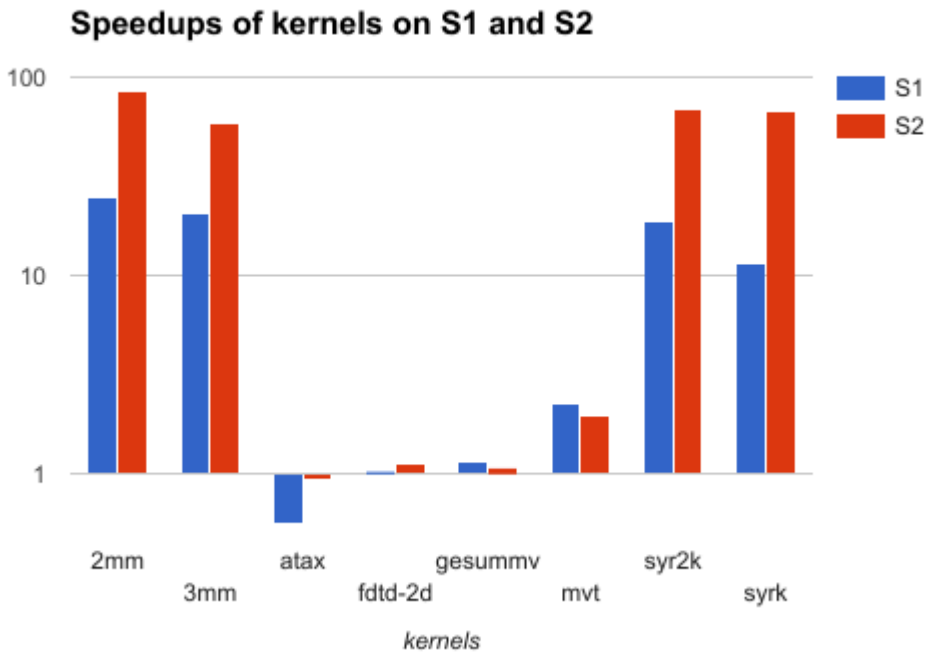
Preliminary Benchmark Results

A kernel's run-time on the CPU (baseline) and the GPU were benchmarked by invoking,

- julia --polly=no --optimize=3 for the CPU, and
- JULIA_LLVM_ARGS="--polly-target=gpu" julia --check-bounds=no for the GPU

The benchmarks were made on 2 systems,

- S1: System with an Intel i7-4820K CPU and a Tesla K20c GPU
- S2: System with an Intel Xeon E5 v5 2620 CPU and a GeForce 1070 GPU



About Me

I'm a final year undergraduate student in computer science and engineering at BITS-Pilani. Hyderabad Campus in India. I'm interested in systems research, specifically in operating systems, parallel programming, compilers and computer architecture for HPC.

I'm currently doing my semester-long undergraduate dissertation at IIT-Hyderabad in polyhedral compilation under Dr. Ramakrishna Upadrasta. It was here that I started working in Polly and took up integrating Polly's GPU targeting capabilities into Julia. Prior to that, I was in touch with one of Polly's lead developers, Dr. Tobias Grosser, and contributed a patch under his guidance.

Earlier, I spent my 2nd year and 3rd year summers at in internship at Indian Institute of Science (IISc), Bangalore and Indian Institute of Technology-Madras(IIT-M), Chennai respectively and programmed GPGPUs. It was at IISc that I first wrote programs for GPUs, specifically simulating neural-networks. This inspired me to work on an idea I had and dedicated my 3rd year summer at IIT-M to it. A poster on this work bagged the Best Poster award at the first edition of the de-HPC conference held at IIT-Bombay in Mumbai. It was also instrumental in landing me in a chance to participate in the Supercomputing 16 conference, SLC, Utah as a part of the "*Experience HPC for Undergraduates*" program.

References

1. [1] Polly — Performing Polyhedral optimization on a low-level intermediate representation
2. [2] Polly-ACC: A heterogeneous compute compiler
3. ppcg's GitHub mirror
4. Polly's GitHub mirror
5. [5]Matthias Reisinger's GSoC 2016 Proposal
6. [6]Matthias Reisinger's GSoC 2016 Final Report