LEGION VIZ

https://github.com/DocSohl/LegionVis

Process book

Milestone 1

Ian Sohl – u0445696

Phil Cutler – u0764757

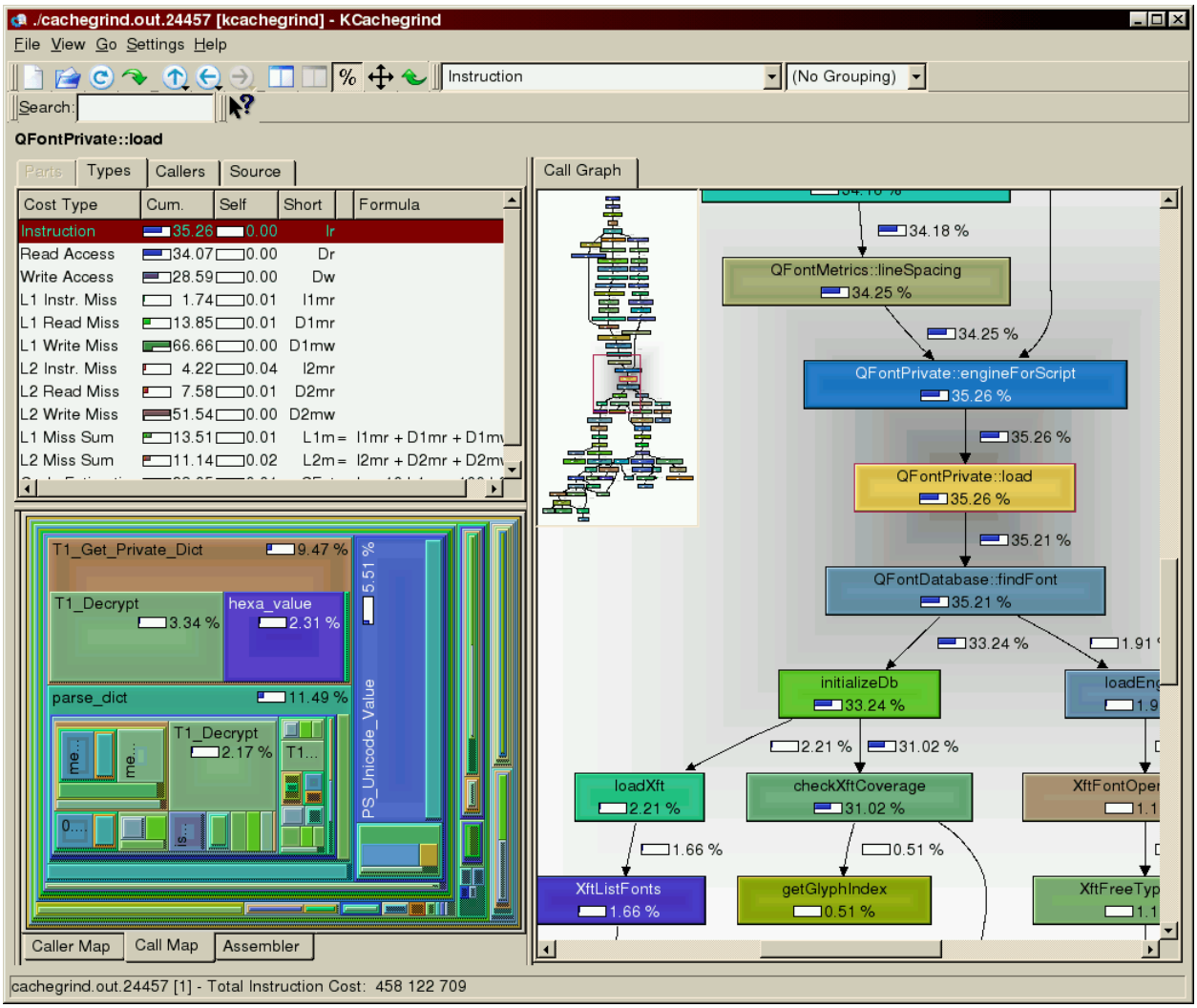
Ariel Herbert-Voss – u0591949

Overview and Motivation

The primary goal of this visualization tool is to allow the user to draw conclusions about the performance of code written in the Legion programming system, which is a data-centric parallel programming system for writing portable high performance programs targeted at distributed heterogeneous architectures. Legion automatically schedules tasks written for high performance applications, meaning that the user doesn't know which processor is running a task or when it starts. Therefore, having a visualization tool that can inform the user of these parameters is very useful for optimizing performance and resources.

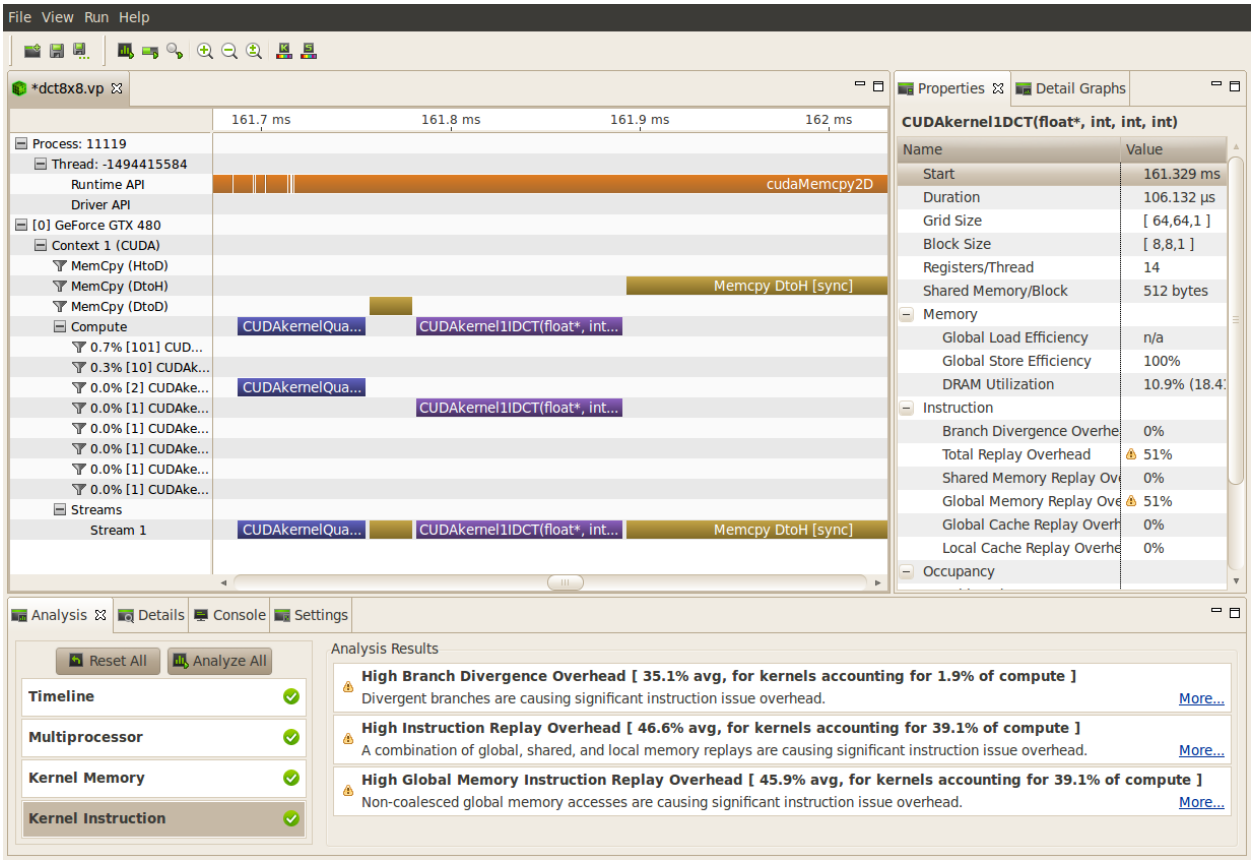
Related Work

Profiler visualization tools are not a new concept and have existed for at least as long as profilers themselves have existed. Perhaps the most well-known example is KCachegrind, which is the visualization tool for the Valgrind programming tool for memory debugging, leak detection, and profiling. Below is the call graph and call map views for KCachegrind.



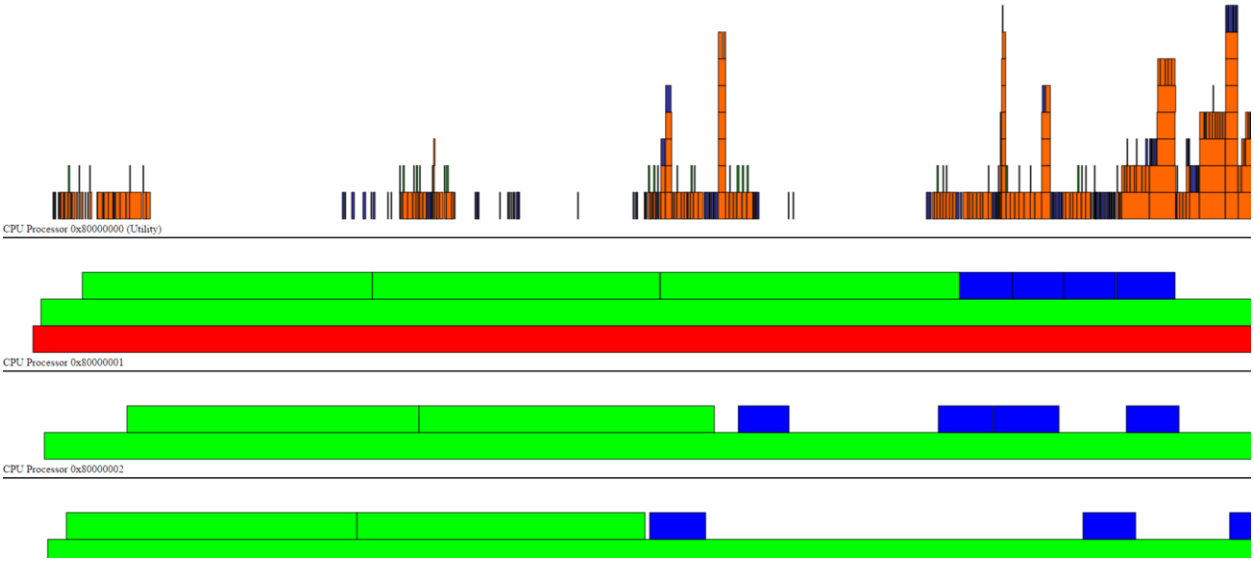
Although not particularly aesthetically-pleasing, KCachegrind is effective in showing the call structure of a particular program and interfaces well with other Linux-based code profilers. However, for highly-parallel code, KCachegrind is inherently insufficient because it relies on the profiled program executing only one thread at a time. Additionally, the tree map graph is ineffective for visualizing call structure in Legion because the user can send data across branches or up the tree due to the unique construction of the system.

With regards to parallel computing, NVIDIA also has a profiler visualization tool for their CUDA programming language. Below is a view of the execution timeline for two host processes in a CUDA kernel.



However, we see that the view is split along function category rather than which processor is running a given function. This sort of tool is not useful for visualizing program execution in Legion because we are unable to identify bottle necks and scaling effects without a proper view of how the processors allocate functions.

Currently, the Legion system has a built in profile visualization tool that produces static images. Below is one such image that shows performance for the first three cores of a 32-core system.



Here, the individual tasks are differentiated by color, rather than a positional axis, as they are less important. This visualization selects for processor parallelism, and task execution time, since tasks in Legion run in an arbitrary ordering and distribution based on the internal scheduler. However, this visualization performs poorly as the number of processors or time increases, which can easily happen in the high performance environments that Legion is intended for.

Questions

Our main question that we want our users to answer with this visualization is whether they have bottlenecks or other major performance issues with their code. A bottleneck can occur when many tasks are waiting upon the completion of a few tasks, thus resulting in irregular load balancing. This load balancing is not only inefficient for the time and computational resources available to the user, but can also have adverse physical effects on very high performance clusters due to the fluctuations in power consumption.

We also want our users to be able to identify how well their programs scale. Since much of computer science and algorithm design uses asymptotic complexity as a metric of performance, our users will want to scale their executions in both data size and processor allocations. We hope that this visualization will allow them to compare separate executions of a single or similar codes and see major differences between them.

Data

Data for this visualization will entirely originate from the end user. We want Legion programmers to be able to upload their own profiling files and visualize them using our system, making the visualization very personal and informative.

These profiling files consist of large plaintext files that detail various operations and actions during the code execution. The primary events that we are interested in are the execution start and stop times of the tasks, as well as which tasks were spawned by other tasks. Data processing happens on both the client and server sides. The server will perform the parsing and most computationally expensive

components of data processing, such as scrubbing for task start and stop times. This information is then sent to the client side to perform the visualization-centric calculations, such as ordering on the screen, and computation of histograms.

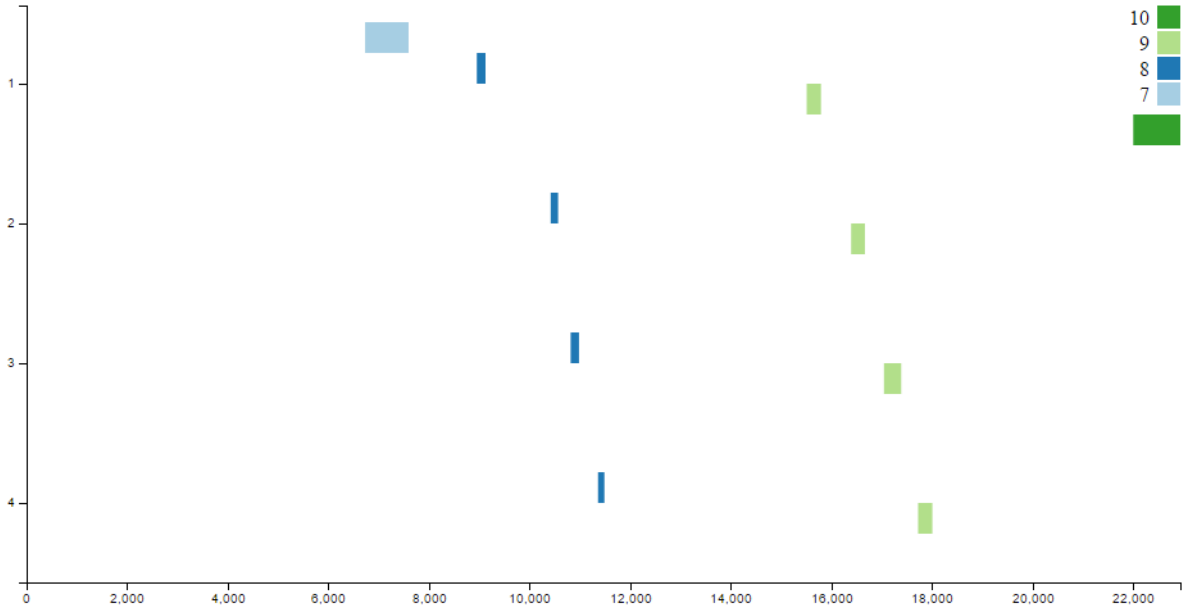
The server will maintain a copy of user submitted files, and allow the user to access them via a unique URL. This allows users to upload the data in one location, and view it from other devices, such as mobile devices, or a coworker's computer for collaboration.

Exploratory Data Analysis

For the initial analysis and selection of test data, we used the built-in visualization offered by Legion. This allowed us to select datasets that varied from simple to complex, and compare our visualizations to an established metric. The other advantage of using the packaged Legion system as a baseline is it's handling of less common cases, such as heterogeneous systems.

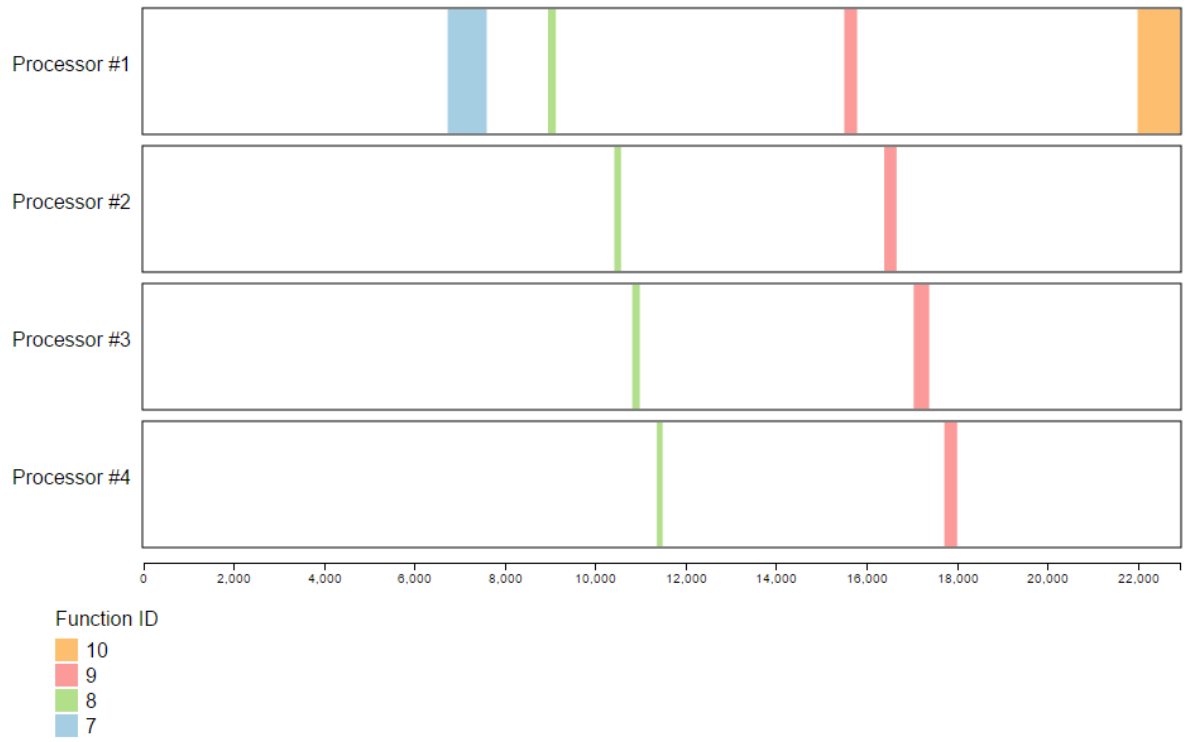
Design Evolution and Implementation

Our first iteration drew very heavily from the Legion default profiler as inspiration.



However, we decided that the spacing between components and color system made it difficult to identify features and relationships in the execution. Additionally, we opted for a more compartmentalized view of different processors rather using the numbers on the y-axis, as that erroneously implied some measure of meaning to the ordering of processors.

Currently, we are on our second iteration, seen below.



This design allows the user to see individual processor lines much more clearly, and features a more useful and intuitive color scale. It also utilizes interactivity by providing tooltips upon mouseover of each task. This tooltip provides the ID of the function, and we plan to provide a persistent properties box that shows details about the task when the user clicks. This visualization also features zooming and panning along the x-axis using the mousewheel and by dragging with the mouse button, respectively.

Evaluation

While we believe that we have a good start for our visualization system, there are still many major improvements that are currently in progress. Our next major goal is to complete a histogram that is drawn by the user brushing across the view with the mouse. This histogram allows the visualization to scale with extreme processor counts by showing the distribution of tasks at a particular point in time, as indicated by the user. This allows the user to temporally identify trends in their execution data by scrubbing across the horizontal axis.

The other major visualization component that we wish to add is a second view that represents the execution structure of the program. Similar to the aforementioned KCachegrind tool, Legion programs can be visualized in a directed acyclic graph in the shape of a tree. Although serial programs are forced to be in a strict tree structure due to the execution order, Legion programs can intelligently move information between tasks, but are spawned in a tree structure. This tool would allow users to evaluate and communicate the structure of their program in a manner not currently available for Legion.