In-Situ Profiling Feedback for GPGPU Code

Armin Pauli, Max von Buelow and Daniel Ströter

Technical University of Darmstadt, Germany

Abstract

The evolving landscape of software development increasingly prioritizes functionality, maintainability, and developer productivity. This typically comes hand in hand with the shortcoming that less focus is invested on optimizing for runtime performance of programs. However, optimizing for performance is an important task in time-critical domains. Additionally, optimizing for performance can be an important way of reducing actual hardware requirements and achieving a better ecological footprint. So, why not bringing program optimization closer to the software engineer and reducing the disconnect between profiling results and their interpretability? This poster presents a GPU-focused in-situ profiling approach that visualizes memory profiling metrics directly inside the source code and gives the software engineer an direct hint for identifying inefficient parts during development. Performance metrics evaluated on each line are highlighted in the source code.

CCS Concepts

• ***Software and its engineering*** → ***Massively parallel systems;*** • ***General and reference*** → ***Performance;*** • ***Human-centered computing*** → *Visualization toolkits;*

1. Introduction

HARWARD, IRWIN, and CHURCHER [HIC10] implemented the idea of visualizing profiling metrics in source code using simple techniques that display the metrics directly in the code. They also added a secondary visualization layer at the beginning of each line to display an additional metric. BECK, MOSELER, DIEHL, and REY [BMDR13] emphasized the psychological perspective of providing in-situ profiling metrics. The study evaluates how profiling data is cognitively processed and concludes that in-situ visualizations physically integrate different representations. This integration mitigates the split-attention effect and reduces extraneous cognitive load, resulting in enhanced cognitive resources for information processing and improved user information processing. The toolset of CITO, LEITNER, BOSSHARD, et al. [CLB*18] offers advanced techniques for interactive visualization in an integrated development environment. While existing works provide a solid theoretical and practical foundation, they are tied to specific programming languages like Java or to CPU hardware. This poster presents an approach capable of inspecting arbitrary source code that can be executed on an NVIDIA GPU and can be easily migrated to other GPU vendors. The presented approach avoids such limitations.

In summary, our contribution is an in-situ code profiling implementation that maps memory profiling metrics to the domain of the source code focused on GPU hardware by estimating per-line metrics of the program.

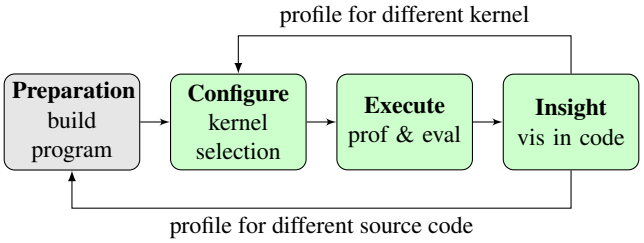

Figure 1: *Flowchart for user interaction using in-situ profiling.*

2. In-Situ Code Profiling

The system's primary interface for user interaction is a Visual Studio Code extension. This choice was made due to this editor's widespread popularity, ease of extension management, and the availability of programming interfaces that facilitate profiling visualization, such as the built-in CodeLens API and code highlighting decorator. Further details on these aspects are discussed in a later section.

Figure 1 shows the workflow that extends the original software engineering process of repeatedly coding, building, and executing the program. The extension includes building the executable, configuring the integrated development environment (IDE) extension for the specific kernel and executable, and starting the profiling process. Once the profiling is complete, the data is evaluated, fed back into the IDE, and made available for analysis. The user can choose to restart the process by selecting the previously profiled kernel af-

ter modifying the source code or by profiling another kernel to view alongside the existing data.

Starting with the *preparation*, users build the source code with debug information, which provides line-number information for the generated device code. This information is essential for later mapping the instrumented memory instructions to understand which lines of code were involved.

Regarding *configuration*, after building the executable, the user can profile one kernel at a time. To do so, the executable containing the kernel source compilation must be specified. Once the extension is enabled in the user's IDE, a new context menu entry becomes available. By right-clicking on the line with the function signature, users can select a menu item to invoke in-situ profiling. The user's selection of the executable automatically determines the Kernel. The profiling process' terminal output will appear in a text field within the window.

The profiling itself is based on a framework [BGF22] that simulates critical parts of the GPU memory pipeline depending on dynamic binary instrumentation techniques. The simulation step is necessary because native profiling metrics are recorded at a per-kernel granularity. This framework annotates cache hit estimates at the granularity of each memory access, which we accumulate by averaging over each line of source code from the debug information.

The profiling process *executes* in the background. Once profiling is complete, the user will receive a prompt to review the data within the source code location. Here, lines of code that generate read or save statements are highlighted with colors based on linear interpolation of memory accesses relative to cache hit rates.

Software developers can repeat the profiling process to verify potential optimizations. They can choose a different kernel or profile the same kernel after making code changes and recompiling. New data is added to their source code tabs, with the old data remaining accessible until a cleanup operation is performed. Cleanup can be initiated through a context menu entry.

3. Results and Conclusion

To demonstrate the profiler's performance, we chose an example program that is expected to have poor caching rates at certain positions in source code. This way, we can explore its behavior in case of poor cache. Such a behavior can occur when threads within a parallel execution unit access memory regions with a high offset without a chance of being merged into a single transaction [Har13]. To simulate this, we include a large array in the kernel code that generates large spaces for each thread that exceed the cache sizes.

As anticipated, the source code displays red lines, with line 29 being particularly intense in color, as shown in fig. 2. A low cache hit rate of 7 % is observed on this line. In contrast, line 27 maintains a cache hit rate of 90 %, which is clearly distinguishable from the other colors. Furthermore, the figure illustrates that our toolset presents additional memory-related metrics in a separate information box. In summary, the visualization effectively highlights areas with low cache hit rates, as indicated by the prominent red color,

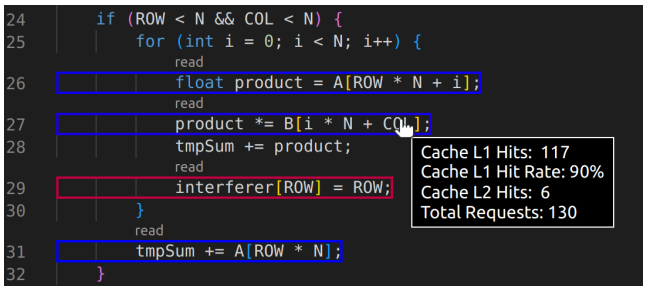

Figure 2: *The kernel for matrix multiplication has been modified resulting in poor caching behavior. This is particularly evident in line* 29*, which is highlighted in an eye-catching red color that stands out from the blue and purple hues.*

which contrasts well with the blue and purple hues. The annotations above the corresponding lines provide clear information for read or store operations.

This poster presents a profiling tool that offers detailed in-situ insights into memory accesses, with a focus on caching behavior at the level of individual lines of code. The tool's insights are evaluated and seamlessly integrated into the Visual Studio Code IDE workflow. Users can conveniently examine profiling data through visualizations and obtain precise numerical values directly in the source code.

Source Code We release source code upon publication.

References

[BGF22] BUELOW, MAX VON, GUTHE, STEFAN, and FELLNER, DIETER W. "FineâGrained Memory Profiling of GPGPU Kernels". *Computer Graphics Forum* 41.7 (Oct. 2022), 227–235. ISSN: 0167-7055. DOI: 10.1111/cgf.14671 2.

[BMDR13] BECK, FABIAN, MOSELER, OLIVER, DIEHL, STEPHAN, and REY, GUNTER DANIEL. "In situ understanding of performance bottlenecks through visually augmented code". *2013 21st International Conference on Program Comprehension (ICPC)*. 2013 IEEE 21st International Conference on Program Comprehension (ICPC) (San Francisco, CA, USA, May 20–21, 2013). IEEE, May 2013. DOI: 10.1109/icpc.2013.6613834 1.

[CLB*18] CITO, JÜRGEN, LEITNER, PHILIPP, BOSSHARD, CHRISTIAN, et al. "PerformanceHat. augmenting source code with runtime performance traces in the IDE". *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. ICSE '18: 40th International Conference on Software Engineering (Gothenburg Sweden). New York, NY, and USA: ACM, May 27, 2018. DOI: 10.1145/3183440.3183481 1.

[Har13] HARRIS, MARK. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. Jan. 2013. URL: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/ 2.

[HIC10] HARWARD, MATTHEW, IRWIN, WARWICK, and CHURCHER, NEVILLE. "In Situ Software Visualisation". *2010 21st Australian Software Engineering Conference*. 2010 21st Australian Software Engineering Conference (Aukland, New Zealand, Apr. 6–9, 2010). IEEE, 2010. DOI: 10.1109/aswec.2010.18 1.