Understanding the Performance of GPGPU Applications from a Data-Centric View

Hui Zhang w.hzhang86@samsung.com
Jeffrey K. Hollingsworth hollings@umd.edu

Motivation

- It's hard for programmers to write efficient code on highly parallel and heterogeneous architectures
- There are few performance tools for CUDA users that can locate inefficient *source code* and guide *user-level* optimizations
- Traditional Code-centric profiling approach is insufficient in investigating data placement issue

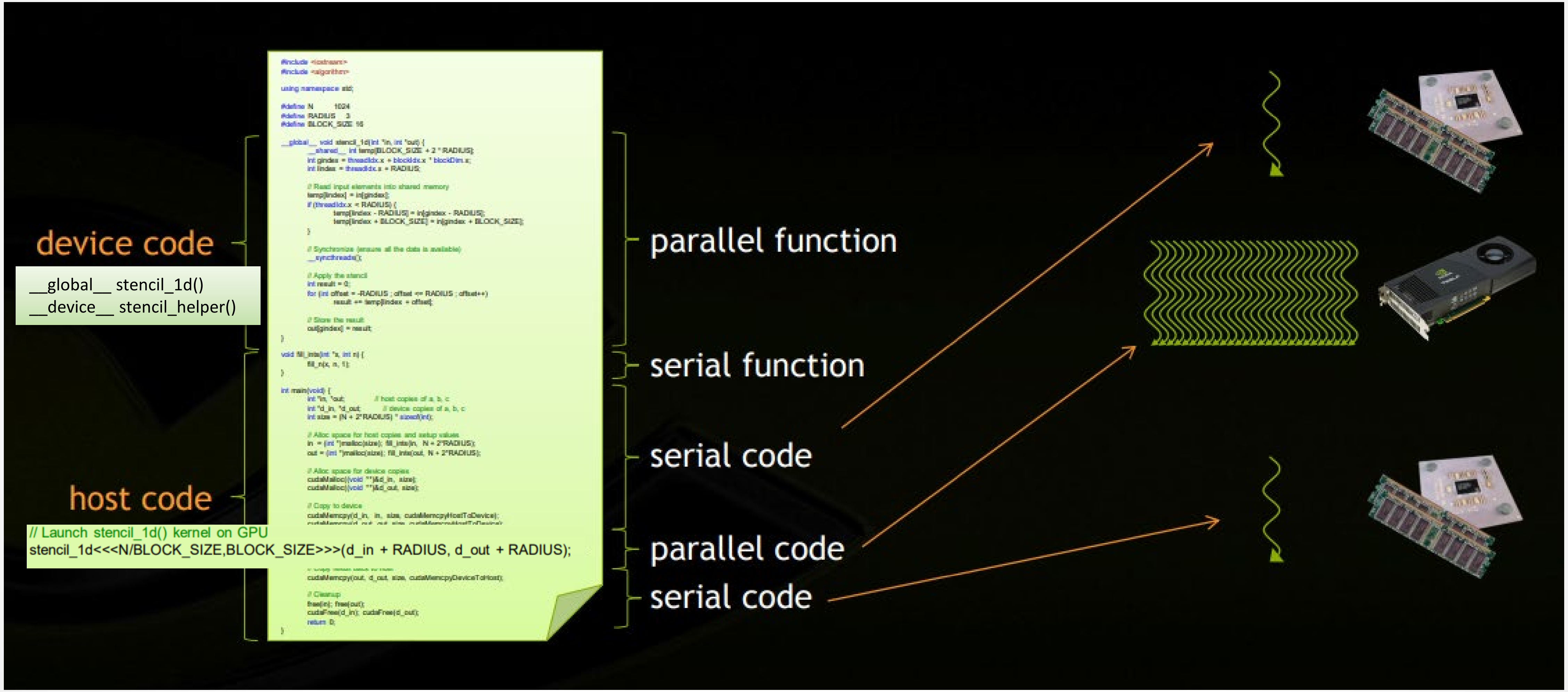
Contributions

- First, the tool offers **fine-grained**, in-depth performance analysis into the kernel execution, providing programmers with finer insight into the GPU kernel execution.

- Second, the tool uses a **data-centric** performance analysis technique to map performance data back to variables in the source code.

- Third, it proposes a method to get the **complete calling context** profiling, including the CPU call stack before a kernel is launched and the GPU call stack within a kernel.

CUDA Programming Overview



* Picture obtained from Nvidia: https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf

Data-centric Profiling

```
int busy(int *x) {
  // hotspot function
  *x = complex();
  return *x;
}

int main() {
  for (i=0; i<n; i++) {
    A[i] = busy(&B[i]) +
        busy(&C[i-1]) +
        busy(&C[i+1]);
  }
}
```

Code-centric Profiling

main: 100%
busy: 100%
complex: 100%

Data-centric Profiling

A: 100%
B: 33.3%
C: 66.7%

Properly Assign Blame

"I didn't say you were to blame… I said I am blaming you."

CUDABlamer Framework

Static Analysis

- Data flow analysis
- Control flow analysis
- Intra-procedural Blame analysis
- Exit variables analysis

Monitored Execution

- CUPTI Callback API: tag kernel invocation
- Libunwind: CPU stack unwinding
- CUPTI Activity API: GPU kernel sampling

Postmortem Process

- Process runtime information
- Reconstruct CPU&GPU calling context
- Inter-procedural Blame analysis
- Determine Blame attribution vars/funcs

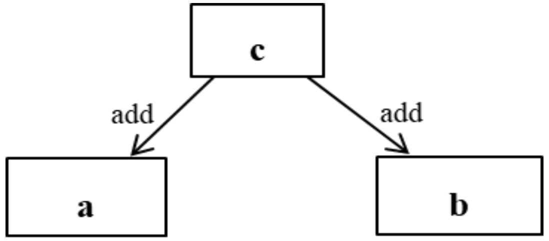
GUI Presentation

- Data-centric profiling result
- Code-centric profiling result

CUDABlamer – Static Analysis

- Graphical Representation to resolve Blame relation

```
var a : int = 6;
var b : int = 7;
var c : int = a + b;
```



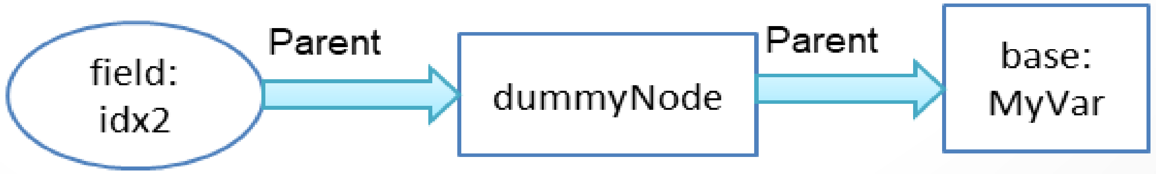
- Resolve LLVM composite instructions to propagate blame hierarchically

%idx1 = getelementptr i32, %struct.MyStruct* %myVar, i64 0, i64 0

(a) Normal GEP instruction

%idx2 = getelementptr i32, {getelementptr i32*,
%struct.MyStruct* %myVar, i64 0, i64 1}, i64 0

(b) Composite GEP instruction

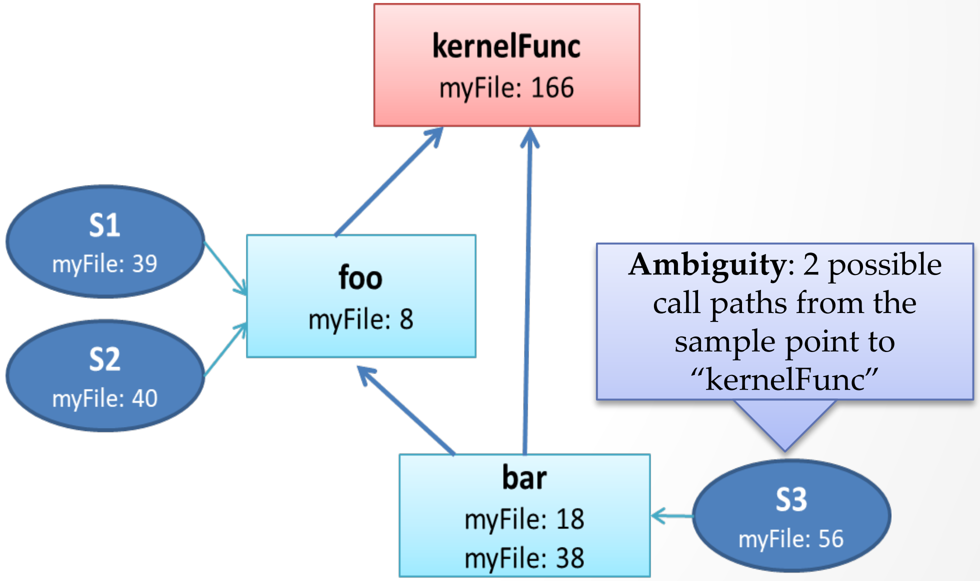


CUDABlamer – Postmortem Process

- Construct Calling Context for CPU-GPU Hybrid Model
  - CPU stack : keep call stack with *Kernel Launch ID (correlationID)*
  - GPU stack for kernel execution: find all paths from sample point to kernel using Depth-First-Search (top & bottom node info from ActivityAPI)
  - Reconstruct the calling context: Connect CPU & GPU stacks through *correlationID*

example

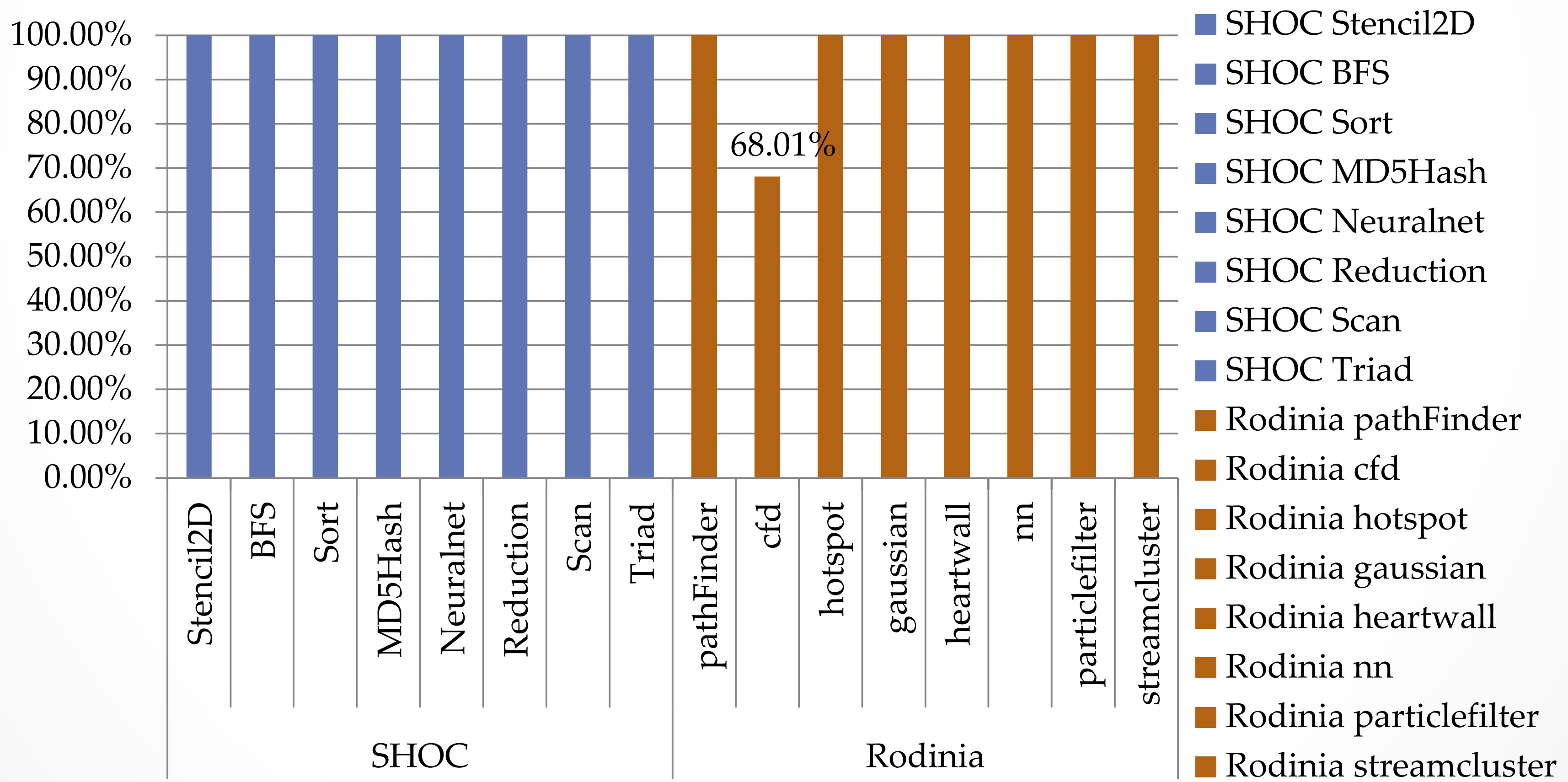
```
1    __global__ void kernelFunc(...){
8       foo(); ...
18      bar(); ...
     }

28   __device__ void foo(){
38      bar();                    ...
39      x = 1;  ...               //Sample 1
40      y = 2; ...                //Sample 2
     }

48   __device__ void bar(){
56      A[i] = B[i]*s;          //Sample 3
88   }
```



Precision Evaluation

- Coverage Metric:

$$coverage = \frac{totalNumSamples - numAmbiSamples}{totalNumSamples}$$



Tool Evaluation – Particlefilter

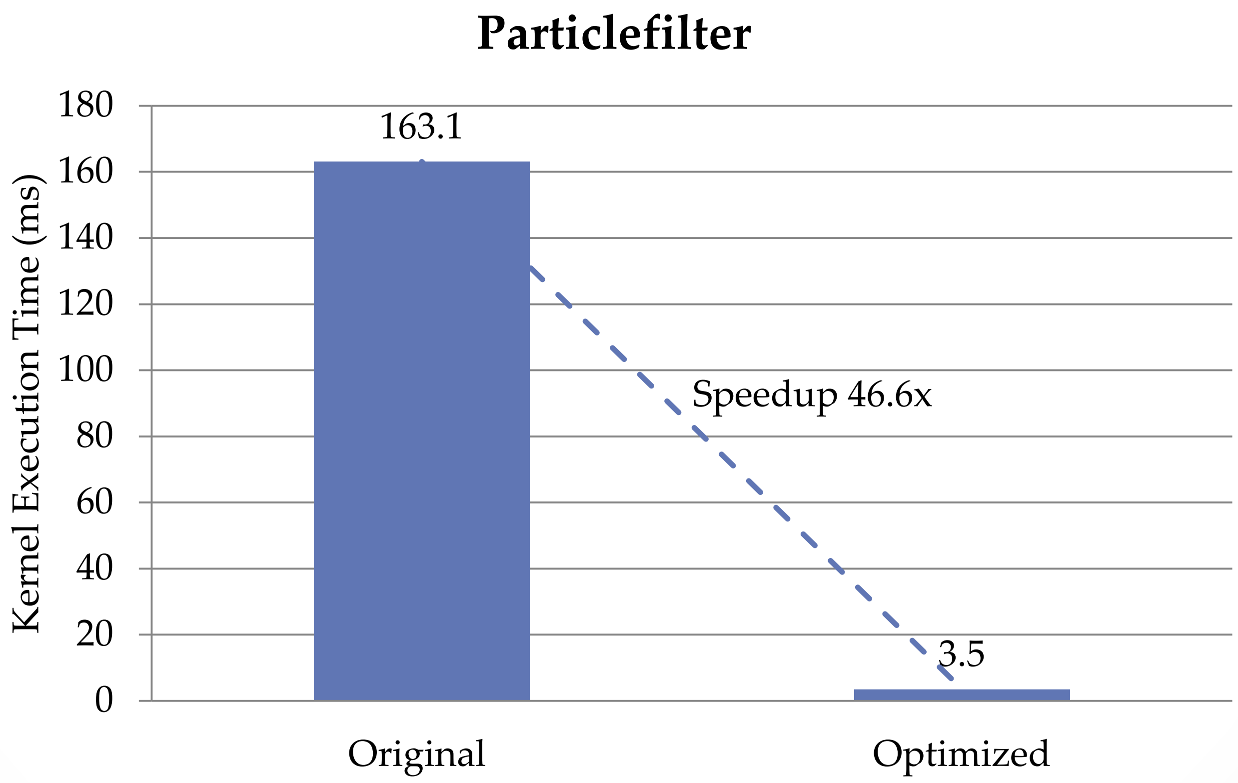
Single-node: 2 NVIDIA Tesla P100 GPUs, each P100 GPU contains **16 GB on-chip** memory and **56 SM** (streaming multiprocessors). Each SM also has **64KB** of **shared** memory. The GPU also provides **48KB** of **constant** memory.

Compilers: nvcc 8.0, gcc 4.8.5 and clang 4.0.1

Variable	Type	Context	Blame
ye/xe	double	main.particleFilter	100%
arrayX/arrayY	*double	main.particleFilter	100%
xj	*double	main.particleFilter	97.9%
yj	*double	main.particleFilter	97.8%
xj_GPU	*double	main.particleFilter	97.9%
yj_GPU	*double	main.particleFilter	97.8%
index	int	main.particleFilter.kernel	95.7%

Tool Evaluation – Particlefilter

- **Optimization**
  - using constant memory for read-only variables *arrayX_GPU, arrayY_GPU, u_GPU, CDF_GPU*



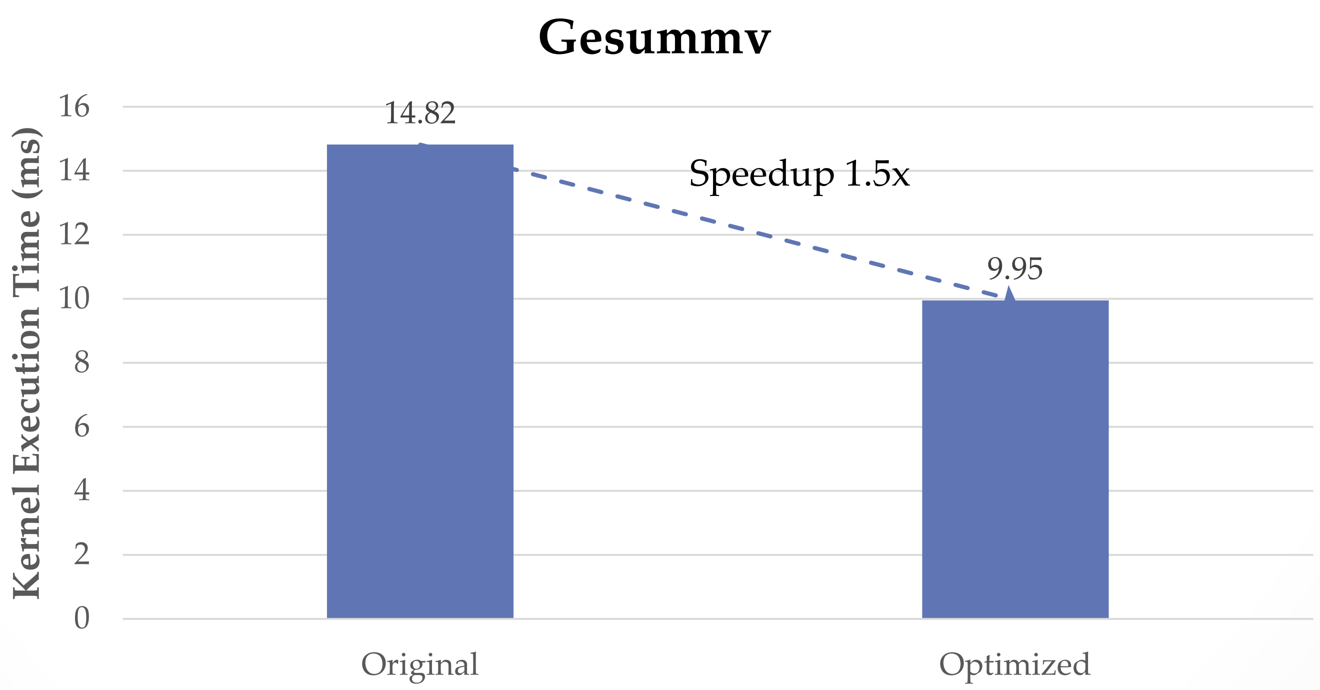
Tool Evaluation - Gesummv

- Gesummv is part of the Polybench test suite and has a kernel that does scalar, vector, and matrix multiplication

Variable	Type	Context	Blame
y_outputFromGpu	*float	main	100%
y_gpu	*float	main.gesummvCuda	100%
tmp_gpu	*float	main.gesummvCuda	52.1%
j	int	gesummv_kernel	4.3%
A_gpu/B_gpu	*float	main.gesummvCuda	1.2%
x_gpu	*float	main.gesummvCuda	1.2%

Tool Evaluation - Gesummv

- **Optimization**
  - *y_gpu* is allocated in the global memory and updating it iteratively is costly. We use temporary variables to hold intermediate result in the *for* loop and assigning the ultimate value to the corresponding array element once in the end



Tool Evaluation - Gramschm

Variable	Type	Context	Blame
A_outputFromGpu	*float	main	99.1%
A_gpu	*float	main.gramschmidtCuda	99.1%
R_gpu	*float	main.gramschmidtCuda	60.6%
nrm	float	main.gramschmidtCuda	19.5%
i	int	Gramschmidt_kernel3	6.7%
Q_gpu	*float	main.gramschmidtCuda	2.8%

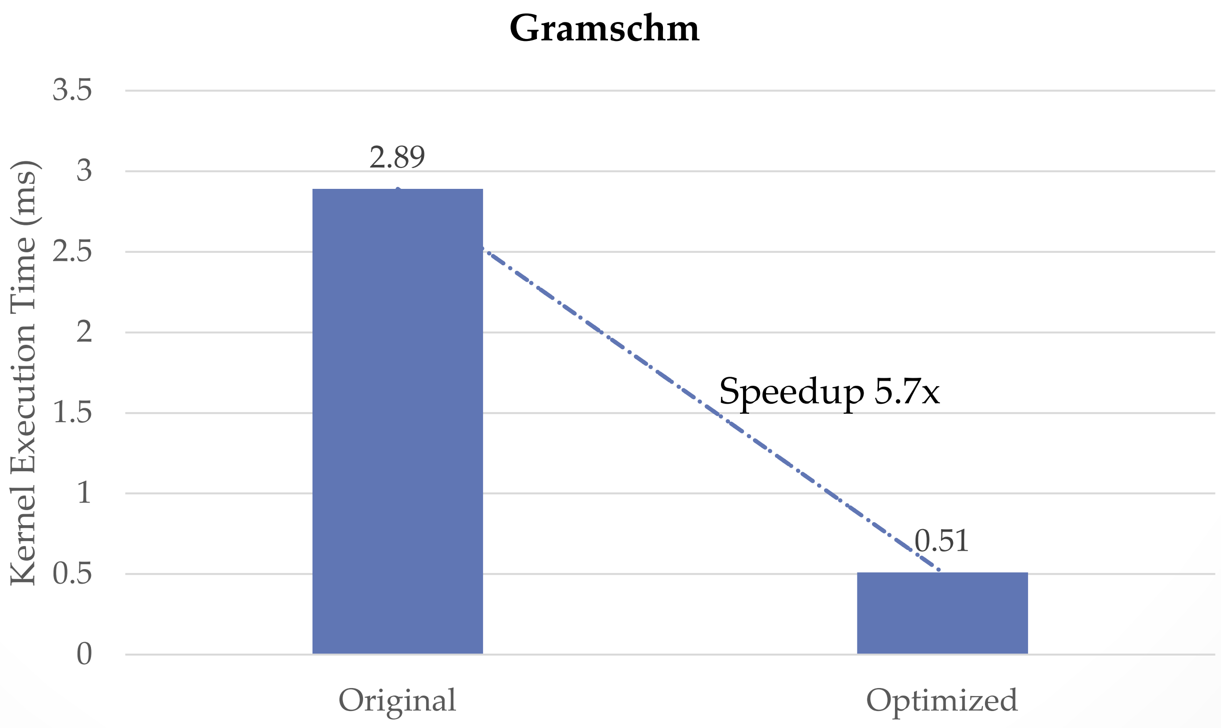
Data-centric

Function	Scope	Blame
main	CPU	100%
gramschmidtCuda	CPU	100%
gramschmidt_kernel3	GPU	78.2%
gramschmidt_kernel1	GPU	19.9%
gramschmidt_kernel2	GPU	1.9%

Code-centric

Tool Evaluation - Gramschm

- **Optimization**
  - *R_gpu*: Use a temporary variable to hold the incremental value of *R_gpu* and do one-time assignment after the loop
  - *Q_gpu*: Use shared memory instead of global memory to store per-block copy of it, and change the column-based access to row-based access



CUDABlamer Overhead

Benchmark name	Clean execution	Static analysis	Monitored execution	Post processing	Runtime overhead	Total overhead
Hotspot	10.43	1.61	10.82	0.83	3.7%	27.0%
Streamcluster	16.96	2.54	115.35	55.46	580%	922%
Particlefilter	10.21	1.34	11.1	1.74	8.7%	38.9%

Unit: seconds

- Static analysis runs once for each benchmark w/ different problem sizes
- Post processing overhead depends on #samples & #blame variables/sample
- Runtime overhead = (Monitored execution / Clean execution) - 1
- Total overhead = (Total profiling time / Clean execution) - 1

- **Runtime overhead** can be high due to the poor performance of CUPTI library provided by NVIDIA when using **PC_SAMPLING** mechanism

Conclusion

- New Performance Attribution for Emerging Programming Models
  - Developed a data-centric CUDA profiler: CUDABlamer
- Complete User-level Calling Context
  - Using static and runtime information to interpolate the complete calling context for heterogeneous architecture
- Valuable Performance Insights
  - Manual optimization gained speedup up to 47x for selected CUDA kernels