NeoRodinia: Evaluation of High-Level Parallel Programming Models and Compiler Transformation for GPU Offloading

Xinyao Yi, Anjia Wang, Yonghong Yan

Department of Computer Science | UNC Charlotte | https://passlab.github.io

Introduction

NeoRodinia is an extensive benchmark suite that evolved from the Rodinia benchmark suite, encompassing 23 real-world applications and 5 microbenchmarks. It addresses the limitations of Rodinia by optimizing OpenMP GPU offloading programs and introducing OpenACC variants.

The main contributions of NeoRodinia include:

- Added missing OpenMP offloading programs and optimized existing ones.
- Added the OpenACC variants.
- The evaluation including performance assessments on various programming models using various compilers, measuring execution time and memory usage.
- These evaluations offer valuable insights into parallel programming models and compiler selection.
- NeoRodinia can be used to guide optimization efforts and help developers, especially beginners to make informed decisions.

We also introduced a 3-tier optimization model, each tier with specific objectives and methods, demonstrated using microbenchmarks. This model not only aims to guide performance programming but also to standardize and improve traceability in optimization processes.

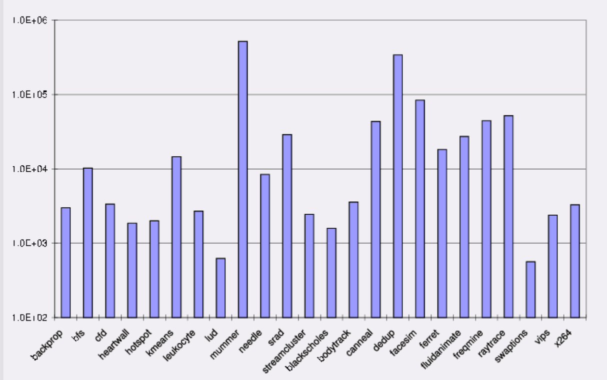
Background & Motivation

Rodinia Benchmark Suite

The Rodinia benchmark suite includes 23 diverse applications and kernels for multi-core CPU and GPU platforms. It covers fields like medical image processing, bioinformatics, and fluid dynamics, implemented in parallel languages like CUDA, OpenCL, and OpenMP. This suite is valuable for evaluating hardware efficiency and testing compilers that support heterogeneous architectures. Several studies have used the Rodinia benchmark suite for performance evaluations.

Issues in Rodinia

- OpenMP variant
  - most programs only have CPU versions
  - existing GPU versions are not optimized
- OpenACC variant
  - no official version
  - the existing unofficial version barely works
- Lack of compiler support evaluation
  - Varied performance across different compilers
  - potentially causing unpredictable behavior



Che S, Sheaffer J W, Boyer M, et al. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads[C]//IEEE International Symposium on Workload Characterization (IISWC'10). IEEE, 2010: 1-11.

Objectives

- Optimize and expand Rodinia
  - the missing OpenMP Offloading version
  - the OpenACC GPU Offloading version
- Serve as a platform for evaluating various compilers
- Provide educational resources
  - showcases optimization processes and various techniques
  - added five user-friendly microbenchmarks

Benchmark Summary

Applications	Dwarves	Domains	CUDA	OpenMP-CPU	OpenMP-GPU	OpenACC
Leukocyte	Structured Grid	Medical Imaging	Existed in Rodinia	Existed in Rodinia	New Added	New Added
Heart Wall	Structured Grid	Medical Imaging	Existed in Rodinia	Existed in Rodinia	New Added	New Added
MUMmerGPU	Graph Traversal	Bioinformatics	Existed in Rodinia	Existed in Rodinia	New Added	New Added
CFD Soiver	Unstructured Grid	Fluid Dynamics	Existed in Rodinia	Existed in Rodinia	New Added	New Added
LU Decomposition	Dense Linear Algebra	Linear Algebra	Existed in Rodinia	Existed in Rodinia	New Added	New Added
HotSpot	Structured Grid	Physics Simulation	Existed in Rodinia	Existed in Rodinia	New Added	New Added
Back Propagation	Unstructured Grid	Pattern Recognition	Existed in Rodinia	Existed in Rodinia	New Added	New Added
Needleman-Wunsch	Dynamic Programming	Bioinformatics	Existed in Rodinia	Existed in Rodinia	New Added	New Added
Kmeans	Dense Linear Algebra	Data Mining	Existed in Rodinia	Existed in Rodinia	New Added	New Added
Bradth-First Search	Graph Traversal	Graph Algorithms	Existed in Rodinia	Existed in Rodinia	New Added	New Added
SRAD	Structured Grid	IImage Processing	Existed in Rodinia	Existed in Rodinia	New Added	New Added
Streamcluster	Dense Linear Algebra	Data Mining	Existed in Rodinia	Existed in Rodinia	New Added	New Added
Particle Filter	Structured Grid	Medical Imaging	Existed in Rodinia	Existed in Rodinia	New Added	New Added
PathFinder	Dynamic Programming	Grid Traversal	Existed in Rodinia	Existed in Rodinia	New Added	New Added
Gaussian Elimination	Dense Linear Algebra	Linear Algebra	Existed in Rodinia	New Added	New Added	New Added
k-Nearest Neighbors	Dense Linear Algebra	Data Mining	Existed in Rodinia	Existed in Rodinia	New Added	New Added
LavaMD	N-Body	Molecular Dynamics	Existed in Rodinia	Existed in Rodinia	New Added	New Added
Myocyte	Structured Grid	Biological Simulation	Existed in Rodinia	Existed in Rodinia	New Added	New Added
B+ Tree	Graph Traversal	Search	Existed in Rodinia	Existed in Rodinia	New Added	New Added
GPUDWT	Spectral Method	IImage/Video Compression	Existed in Rodinia	New Added	New Added	New Added
Hybrid Sort	Soring	Sorting Algorithms	Existed in Rodinia	New Added	New Added	New Added
Hotspot3D	Structured Grid	Physics Simulation	Existed in Rodinia	Existed in Rodinia	New Added	New Added
Huffman	Finite State Machine	Lossless data compressiol	Existed in Rodinia	New Added	New Added	New Added
AXPY	Dense Linear Algebra	Linear Algebra	New Added	New Added	New Added	New Added
Mat-Vec Mul	Dense Linear Algebra	Linear Algebra	New Added	New Added	New Added	New Added
Mat-Mat Mul	Dense Linear Algebra	Linear Algebra	New Added	New Added	New Added	New Added
Sum	Dense Linear Algebra	Linear Algebra	New Added	New Added	New Added	New Added
Stencil	Dense Linear Algebra	Linear Algebra	New Added	New Added	New Added	New Added

Code Optimization Example

Stream Cluster - OpenMP

```
#pragma omp target teams distribute parallel for
    map(to: d_points[0:chunksize], center_table[0:chunksize])
    map(tofrom:switch_membership[0:chunksize], lower[0: stride*(nproc+1)])
    num_teams(1024) num_threads(512) reduction(+: cost_of_opening_x)
    map(to: d_coord[0:num * dim])
for (int i = 0; i < num_points; i++) {
  float x_cost = d_dist(i, x, dim, d_coord) * points->p[i].weight;
  float current_cost = d_points[i].cost;
  if (x_cost < current_cost) {
    switch_membership[i] = 1;
    cost_of_opening_x+=x_cost-current_cost;
  } else {
    int assign = d_points[i].assign;
  #pragma omp critical
    lower[center_table[assign]] += current_cost - x_cost;
  }
}
```

```
#pragma omp target data map(allo
{ ...
   // only if necessary
#pragma omp target update to(d_coord[0:num*dim]) while(!center_is_stable()){
#pragma omp target teams distribute parallel for
  map (to:d_points [0:chunksize], center_table [0: chunksize])
  map(tofrom:switch_membership [0: chunksize], lower[0:stride * (nproc+1)])
  num_teams(1024) num_threads(512) reduction(+:cost_of_opening_x)
for (int i = 0; i < num_points; i++) {
 ... // compute distance between points
 if (x_cost < current_cost) {
 ...  } else {
    int assign = d_points[i].assi
    #pragma omp atomic
    lower[center_table[assign]] +
  } ... } } }
```

Using function *center_is_stable* to avoid redundant calculations and achieve data reuse.

Atomic directive can combine several hardware instructions into one instruction to execute, thus avoiding the calculation of many intermediate results.

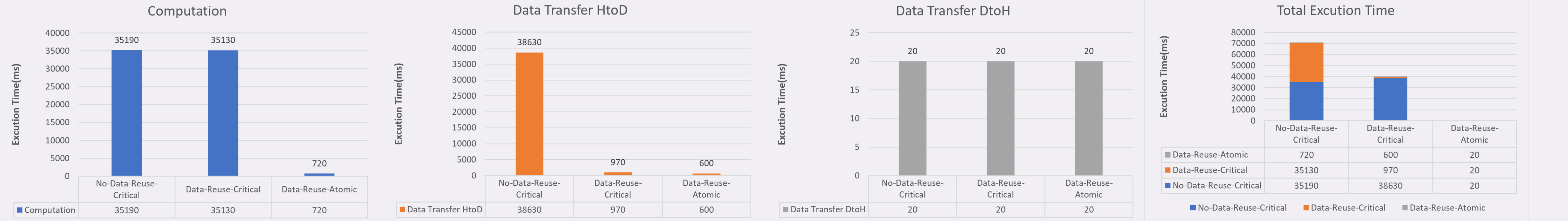
Stream Cluster - OpenACC

```
#pragma acc data create(d_coord [0:
  chunksize * dim])
{ ... // only if necessary
#pragma acc update device(d_coord[0:num *
  dim])
  while (!center_is_stable()) {
#pragma acc parallel loop copyin(d_points
  [0: chunksize], center_table[0:chunksize])
  copy(switch_membership [0:chunksize],
  lower [0: stride * (nproc + 1)])
  num_gangs(1024) num_workers(1)
  vector_length(512) reduction(+ :
  cost_of_opening_x) present(d_coord)
for (int i = 0; i < num_points; i++) {
  ... // compute distance between points
  if (x_cost < current_cost) {
    ...
  } else {
    int assign = d_points[i].assign;
    #pragma acc atomic update
    lower[center_table[assign]] +=
current_cost - x_cost;
  }
... } } }
```

Use the specific OpenACC directives instead of directly using the simpler *kernel* directive. The directives we use basically achieve the same functions as using OpenMP.

Preliminary Results

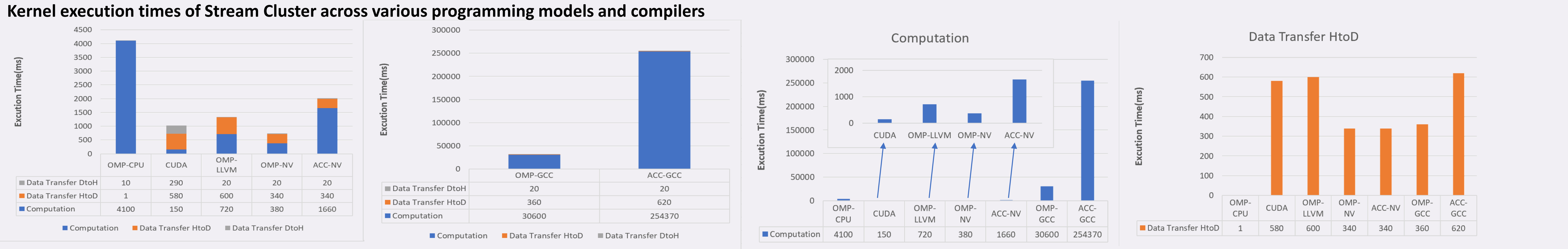
Kernel execution times of stream cluster (compiled by LLVM, input size: 64k.)



(a) computation time (b) time for data transfer from host to device (c) time for data transfer from device to host (d) total execution time

The experimental results presented compare three versions of OpenMP: an initial version without optimization, a version using conditional statements to enable data reuse, and a version utilizing atomic directives instead of critical directives. The results demonstrate that data multiplexing significantly reduces the time required for data transmission, while the use of atomic directives leads to a reduction in computing time.

Kernel execution times of Stream Cluster across various programming models and compilers



(a) total execution time for LLVM, NVCC and NVC compilers (b) total execution time for gcc compiler (c) computation time (d) time for data transfer from device to host

This figure presents a comparison of different models and compilers. The results indicate that the CUDA version achieves the highest calculation efficiency, but incurs significant data transmission overhead. The OpenMP offloading version compiled with nvc delivers the best overall performance. Another interesting finding is we basically achieve the same functions in OpenACC with OpenMP, but due to the limitations of OpenACC compiler and runtime support, we cannot achieve the same performance as using OpenMP, which also reflects one of the important functions of benchmarks.

3-Tier Optimization Model

Matrix-Vector Multiplication

```
void matvec_P1(int N, REAL *A, REAL *B, REAL *C) {
    REAL temp;
    #pragma omp parallel for shared(N,A,B,C) private(i,j,temp)
    for (int i = 0; i < N; i++) {
        temp = 0.0;
        for (int j = 0; j < N; j++)
            temp += A[i * N + j] * B[j];
        C[i] = temp;
    }
}
```

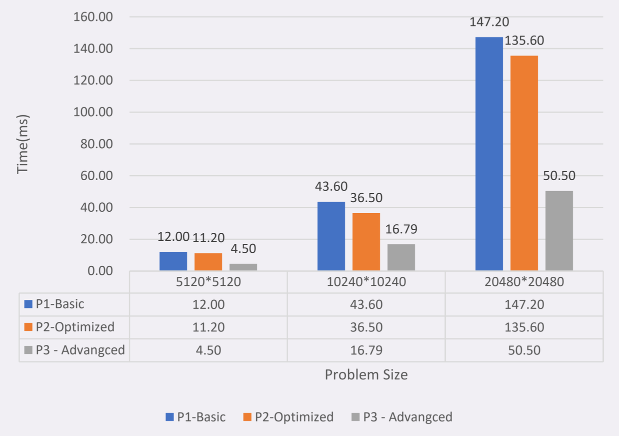
Basic Version

```
void matvec_P2(int N, REAL *A, REAL *B, REAL *C) {
    REAL temp;
    #pragma omp parallel shared(N,A,B,C) private(i,j,temp)
    #pragma omp for schedule(guided, 64)
    for (int i = 0; i < N; i++) {
        temp = 0.0;
        for (int j = 0; j < N; j++)
            temp += A[i * N + j] * B[j];
        C[i] = temp;
    }
}
```

Optimization - Scheduling

```
void matvec_P3(int N, REAL *A, REAL *B, REAL *C) {
    REAL temp;
    #pragma omp parallel shared(N, A, B, C) private(i, j, temp)
    {
        #pragma omp for schedule(guided, 64)
        for (int i = 0; i < N; i++) {
            temp = 0.0;
            #pragma omp simd reduction(+:temp)
            for (int j = 0; j < N; j++) {
                temp += A[i * N + j] * B[j];
            }
            C[i] = temp;
        }
} }
```

Optimization - Vectorization



We have introduced an optimization model with three tiers, denoted as P1, P2, and P3, where 'P' stands for parallel. Each tier is distinguished by its specific optimization objectives and methodologies. We use five microbenchmarks to showcase this model. The primary aim of designing this model is to provide programmers, particularly beginners, with an understanding of how the optimization process unfolds. Additionally, we aim to standardize some optimization steps, enabling a more structured and traceable optimization procedure.

- The P1 level is basic optimization, with the goal of balancing the load and minimizing memory access.
- The P2 level represents an advanced phase of optimization efforts including optimizing scheduling, block size, selection, and both static and dynamic scheduling.
- The P3 level represents the most challenging optimization strategy, including tiling techniques, vectorization and sometimes requiring manual modifications to the code.

The experimental results are consistent with our expectations. As we delve deeper into optimizations, performance continues to improve progressively.

Contact Us

E-mails:

Xinyao Yi xyi2@uncc.edu Anjia Wang awang15@uncc.edu

Yonghong Yan yyan7@uncc.edu

HPCAS Lab:

Computer Science Department

University of North Carolina at Charlotte, Charlotte, NC, USA