Kernel Fusion/Decomposition For Automatic GPU-Offloading
Alok Mishra¹, ², Martin Kong², Barbara Chapman¹, ²
¹Stony Brook University – USA, ²Brookhaven National Laboratory – USA

OpenMP
• More portable
• Comparatively easier to code
• Making the code parallel and handling data is still the responsibility of scientists.

Proposed
• Design & build a compiler framework that can automatically offload profitable regions of code to GPU
• Parallel regions detection
  – Loops, functions, etc.
  – Patterns / Data Analysis
  – Evaluate various scenarios resulting from fusion/decomposition kernels
• Novel and adaptive cost model
• Profitability of offloading each kernel variant
• Code Generation
• Insert pertinent OpenMP code

High Level Flow

In the backend we use LLVM/Chang tools. Analyze Pass identifies all possible kernels in the code and suggests various variants for offloading. Cost Model statically compares the potential performance amongst the various kernels generated. Kernel Transformation adds pertinent OpenMP code to the kernel to support offloading.

Targeted Code Examples

Input Code 1 – Multiplication of 3 matrices A, B and C using an inter-temporal partition C1 to get the final resultant D.

```c
#pragma omp parallel for collapse(2) device(k)
for(int k=0; k<n; k++) {
    for(int i=k*N/n; i<(k+1)*N/n; i++)
        for(int j=0; j<N; j++) {
            C1[i][j] += A[i][k] * B[k][j];
        }
}
```

Output Code 1.1 – Auto-generated code to offload the two kernels to GPU.

```c
# pragma omp target data map(to: A[N][N], B[N][N], C1[N][N])
# pragma omp target teams distribute parallel for collapse(2)
for(int k=0; k<n; k++) {
    for(int i=k*N/n; i<(k+1)*N/n; i++)
        for(int j=0; j<N; j++) {
            D[i][j] += C1[i][k] * C[k][j];
        }
}
```

Decomposition
• Copying a large array. Based upon different scenarios

```
Output Code 1.3 – Auto-generated code from Input Code 1 to reuse same data in multiple kernels. We correctly identify data used to be seed data from the device.

```c
#pragma omp targets data maps: A[0][N], B[0][N], C1[0][N], C[0][N]
#pragma omp target teams distribute parallel for collapse(2)
for(int k=0; k<n; k++) {
    for(int i=k*N/n; i<(k+1)*N/n; i++)
        for(int j=0; j<N; j++) {
            D[i][j] += C1[i][k] * C[k][j];
        }
}
```

Related Tools
• Grid Library
  – Used in Lattice QCD application
• Data parallel C++ math object library
  – Approx. 100k lines of code
  – https://github.com/palohyre/Grid
• Polly, IMPACT ‘11
  – Polyhedral loop optimizer for LLVM
• Data Analysis and reuse opportunity
  – https://polly.llvm.org
• DawnCC, TACO, July ’17
  – Parallelism detection
  – After code to include OpenMP directives
  – http://cuda.de/afnig/da/uren

Criteria
• Fusion
  – CPU: baseline CPU parallel code
  – GPU: target teams distribute parallel
• Collapse: collapse nested loops on GPU
• Fusion 1: in an affine code merge multiple loops to form single kernel
• Fusion 2: based upon proximity, reuse data between different kernels
• Decomposition
  – ’N’ device: based upon the input size, decompose computation to multiple (’N’) devices upon availability
• Cost Models
  – Data Transfer Cost: time spent in data transfer between CPU and GPU
  – Initialization Cost: time to initialize GPUs
• Compute Cost: Build upon existing cost model

Challenges
• Portability
  – Highly dependent on underlying architecture and choice of programming model (e.g. CUDA)
• Programmability
  – Different from existing programming languages
  – Extensive relactoring of code is required
• Parallelism
  – Is the code parallel enough?
• Data Handling
  – Requires explicit data transfers

GPU
• Ideal for heavy computational workload, like Matrix Multiplication, Linear Algebra, etc.