Assignment 09
MC970/MO644 — Introduction to Parallel Programming (1s17)

Marcio M Pereira[a]

[a] *Computer Science Institute, University of Campinas — UNICAMP — Brazil*

Problem Description

PolyBench is a benchmark suite of 30 numerical computations with static control flow, extracted from operations in various application domains (linear algebra computations, image processing, physics simulation, dynamic programming, statistics, etc.).

In this assignment, you will work with Matrix Vector Product and Transpose — mvt, from Linear Algebra. The mvt application is a matrix vector multiplication composed with another matrix vector multiplication but with transposed matrix. It takes the following as inputs:

- A: $N * N$ matrix
- y1, y2: vectors of length N

and gives the following as outputs:

- x1: vector of length N, where $x1 = x1 + Ay1$
- x2: vector of length N, where $x2 = x2 + A^T y2$

Topic Area: OpenMP 4.0 Accelerator Programming Model

Introduction

Although OpenCL provides a library that eases the task of offloading kernels to acellerator devices, its function calls are complex, have many parameters and require the programmer to have some knowledge of the device architecture's features (e.g. block size, memory model, etc.) in order to enable a correct and effective usage of the device. Hence, OpenCL can still be considered a somehow low-level library for heterogeneous computing.

Introduced through OpenMP 4.0 the new *OpenMP Accelerator Model* proposes a number of new clauses aimed at speeding up the task of programming heterogeneous architectures. This model extends the concept of offloading and enables the programmer to use dedicated directives to define offloading target regions that control data movement between host and devices. Although most OpenMP directives used for multicore hosts can also be used inside target regions, the new accelerator model easies the tasks of identifying data-parallel computation.

ACLang is an LLVM/Clang based compiler aimed at implementing the OpenMP Accelerator Model. It adds a new *runtime library* to LLVM/CLang that supports OpenMP offloading to accelerators like GPUs and FGPAs. Kernel functions are extracted from OpenMP annotated regions and are dispatched as OpenCL or SPIR code to be loaded and compiled by OpenCL drivers before being executed by the device.

Listing 1 presents two loops from the `mvt` program of the Polybench benchmark suite. In the first loop the program computes the matrix vector multiplication followed by the transpose between `A` and `y1` storing the result into vector `x1`. The second loop does a similar task for `A`, `y2` and `x2`.

Your first task will be to annotate the runMvt loops with OpenMP 4.X pragmas. You will use the `target` clause to define the portion of the program that will be executed by the device (in the example, GPU). The `map` clause can details the mapping of the data between the host and the target device. For example in the first kernel of Listing 1 inputs (`A` and `y1`) can be mapped *to* the GPU, and array `x1` can mapped *to/from* the GPU. This means that array `x1` is read and written during the kernel execution in the GPU. This strategy offers maximal flexibility to the developer to decide what part of the code is profitable to run on which architecture.

Email address: `mpereira@ic.unicamp.br` (Marcio M Pereira)

Listing 1: Piece of Polybench mvt benchmark application

```
/* Problem size */
#define N 2048  // also test with 4096 and 8192

void init_array(float* A, float* x1, float* x2, float* y1, float* y2)
{
  int i, j;

  for (i = 0; i < N; i++) {
     x1[i] = ((float) i) / N;
     x2[i] = ((float) i + 1) / N;
     y1[i] = ((float) i + 3) / N;
     y2[i] = ((float) i + 4) / N;
     for (j = 0; j < N; j++)
        A[i*N + j] = ((float) i*j) / N;
  }
}

void runMvt(float* a, float* x1, float* x2, float* y1, float* y2)
{
  int i, j;

  for (i=0; i<N; i++) {
     for (j=0; j<N; j++)
        x1[i] += a[i*N + j] * y1[j];
  }

  for (i=0; i<N; i++) {
     for (j=0; j<N; j++)
        x2[i] += a[j*N + i] * y2[j];
  }
}
```

Options that may lead to better performance

ACLang leverages on ISL polyhedral model optimizations to transform the extracted loops so that they can be tiled and mapped to the blocks and threads in the OpenCL code. The *ACLang* polyhedral optimization engine implements a multi-level tiling strategy tailored to the multiple levels of parallelism and to the memory hierarchy of the accelerator. As a example, tiling can be directly applied to the loops of Listing 1, as outermost loops can be executed in parallel because their iterations update disjoint parts of the `x1` and `x2` arrays.

ACLang also provides a vectorization pass developed to exploit the vector instructions available in OpenCL. Kernels are vectorized in a two step process. First, *ACLang* uses the polyhedral otimization engine to re-structure loop nests so as to determine the innermost loops that can be vectorized. Second, vector optimization engine replaces the identified statements with vector instructions.

Your second task is to show detailed speed-ups with respect to sequential code, when using the *ACLang* compiler to generate code for the mvt benchmark. You will use three optimization flavors:

basic kernel generation $ aclang -O3 -opt-poly=none

tiling optimization $ aclang -O3 -opt-poly=tile

vectorization $ aclang -O3 -opt-ploy=vectorize

Problem Sizes

The problem sizes are primarily derived from the memory requirements such that different levels of the memory hierarchy are exercised. For this experimental evaluation, three differents problems sizes to mvt benchmark must be used:

MEDIUM N = 2048

LARGE N = 4096

EXTRALARGE N = 8192

Timing/profiling options

Your thrid task will be the analysis of the OpenCL overhead. To do this, the library was instrumented to collect the time of offloading and kernel execution. To enable this, you must use the argument:

```
$ aclang -O3 -rtl-mode=profile -opt-poly=...
```

Submission

Your submission will consist of a report with the graphs showing the results observed and their analysis of the performance due to the size of the data and the overhead that OpenCL has in the computation itself.