pocl: A Performance-Portable OpenCL Implementation

Kaushik Kulkarni

December 5, 2018

Outline

About the work

- Published in 2015
- Still an actively maintained project
    - Source code: `https://github.com/pocl/pocl`
    - Latest release: `v1.2` in September 2018
    - Latest commit $\sim$ 25 days ago.

Overview of the work

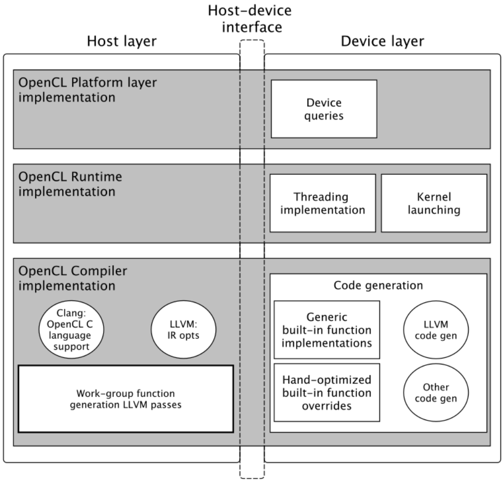
- `OpenCL` codes are platform portable
- However, performance portability is not guaranteed
- `pocl`'s approach: *target-specific* compiler transformations

Performance portability

Variables that affect the performance of an `OpenCL` code

- Address space of variables
- Work group sizes
- Memory access pattern
- Optimizations performed by the kernel compiler of the `OpenCL` implementation

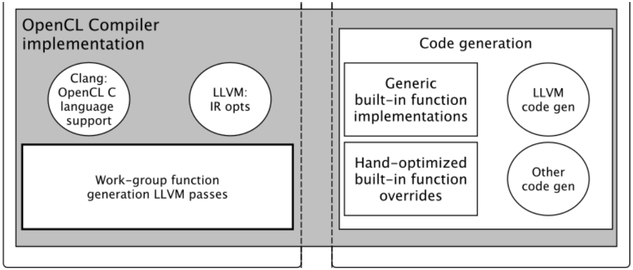
Pieces of an OpenCL implementation



Host-device interface
Host layer
Device layer
OpenCL Platform layer implementation
Device queries
OpenCL Runtime implementation
Threading implementation
Kernel launching
OpenCL Compiler implementation
Clang: OpenCL C language support
LLVM: IR opts
Work-group function generation LLVM passes
Code generation
Generic built-in function implementations
LLVM code gen
Hand-optimized built-in function overrides
Other code gen

Figure: Sub-components of an OpenCL implementation[pg.6]

Kernel compiler


OpenCL Compiler implementation
Clang: OpenCL C language support
LLVM: IR opts
Work-group function generation LLVM passes
Code generation
Generic built-in function implementations
LLVM code gen
Hand-optimized built-in function overrides
Other code gen

`pocl` Compilation chain overview

- The `pocl` kernel compiler is `Clang`-based and generates LLVM IR
- Follows a target based execution model
  - *SIMT* architectures(like GPUs):
    - $\Rightarrow$ Generate code for a single work item
  - Else:
    - $\Rightarrow$ Transform LLVM IR to form multi-work-item work-group functions

pocl Compilation chain overview

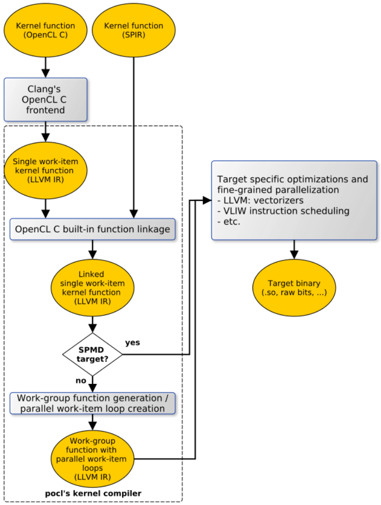

Kernel function (OpenCL C)
Kernel function (SPIR)
Clang's OpenCL C frontend
Single work-item kernel function (LLVM IR)
OpenCL C built-in function linkage
Linked single work-item kernel function (LLVM IR)
SPMD target?
yes
no
Work-group function generation / parallel work-item loop creation
Work-group function with parallel work-item loops (LLVM IR)
pocl's kernel compiler
Target specific optimizations and fine-grained parallelization
- LLVM: vectorizers
- VLIW instruction scheduling
- etc.
Target binary (.so, raw bits, ...)

Figure: Compilation strategies for different backends[pg.12]

LLVM IR

- LLVM IR is a low-level language close to assembly language and abstracts the underlying architecture in a generic manner
  - Data Structures: typed “virtual” registers
  - SSA

Nomenclature

- *Basic blocks:* A group of statements that do not include any conditional statements
- *Control Flow Graphs*: a directed graph with BBs as nodes, arranged in a schedulable manner

Parallel Region in the work group functions

Definition

A collection of statements that must be executed by all the work items in the work group before proceeding to any other statements.

The `OpenCL` specification does not require an ordering of the WIs $\Rightarrow$ *Parallel region is embarrassingly parallel across the WIs*

Generation of Parallel Work-Group Functions

Strategy

1. Identify the parallel region across WIs in the work group functions
2. Annotate the parallel regions of loops across the WIs and with LLVM metadata that serves as directives for the LLVM loop vectorizer

Identifying the parallel region

- The algorithm is categorized for the following cases:
    - No barrier
    - Unconditional barriers
    - Conditional barriers
    - Barriers in loops
- The placement of the barrier in the kernel will affect the parallel region formation in the multi-WI work group functions

Parallel Region Formation: No barrier

- Identifying parallel region is trivial for a kernel with no barriers(*or a CFG with just one BB*) i.e. the entire kernel is the parallel region

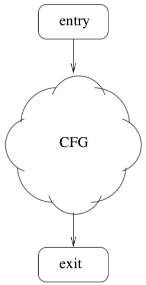


Figure: Single WI function

$\longrightarrow$

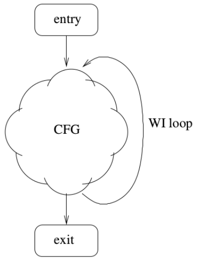


Figure: Mutliple WI-work group function[pg. 15]

OpenCL specification for barriers

Unconditional barrier

- node *A* is said to dominate node B in a CFG if every path from the entry to node *B* has to go through node *A*
- A barrier *B* is said to be an unconditional barrier if the barrier dominates the *'exit'* node

Example

```
__kernel void uncond_barrier_kernel(__global float *a)
{
  int gid = get_global_id(0);
  a[gid] = 2*a[gid];
  barrier(CLK_GLOBAL_MEM_FENCE);
  a[gid] = 2*a[gid];
}
```

OpenCL specification for barriers

Conditional barrier

- If a work item executes a conditional barrier then all the other WIs must also execute the barrier
    - *Same rule applies for barriers in a loop*

Non-example: *Kernel with undefined behavior*

```
__kernel void crash_kernel(__global float *a)
{
  int lid = get_local_id(0);
  int gid = get_global_id(0);
  a[gid ] = 2*a[gid];
  if(lid == 0)
    barrier(CLK_GLOBAL_MEM_FENCE);
  a[gid] = 2*a[gid];
}
```

Parallel Region Formation: Unconditional Barrier

- Split the CFG into sub-CFGs between the unconditional barriers and then form the parallel regions from those sub-CFGs[1]

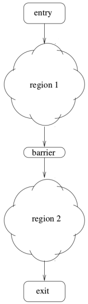


Figure: Single WI function

$\longrightarrow$

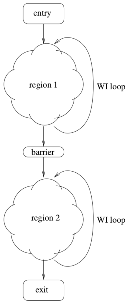


Figure: Mutliple WI-work group function[pg. 15]

[1]One can assume implicit unconditional barriers at the *entry* and *exit* nodes

Parallel Region formation: Conditional barriers

- The CFG will contain atleast one node such that it has more than one predecessor barrier
- $\Rightarrow$ cannot apply the algorithm for unconditional barrier

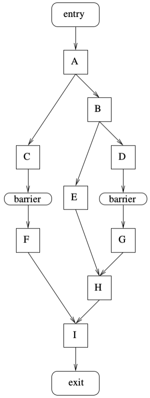
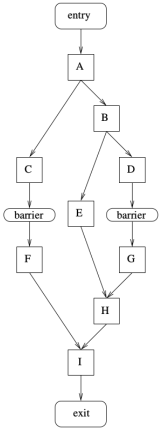


Figure: CFG for a conditional barrier WI[pg.16]

Parallel Region formation: Conditional barriers

- From the `OpenCL` specification for conditional barrier, we can claim that *"All the WIs in a work group take the same path from entry to exit"*
- $\Rightarrow$ by *"tail duplication"* of the sub-CFG between a conditional barrier and the exit node we can apply the algorithm for unconditional barrier to each one of the tails

Parallel Region formation: Conditional barriers



$\longrightarrow$

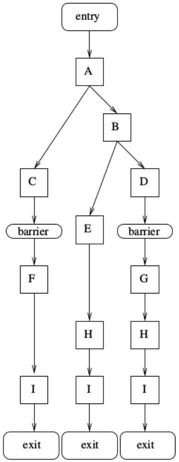
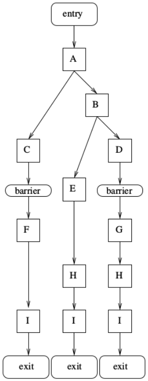


Figure: Single WI function[pg. 16]

Figure: Tail duplicated WI function[pg. 18]

Parallel Region formation: Conditional barriers

- The *"tail-duplicated"* CFG's parallel region can be dealt in the similar way as unconditional barriers



$\longrightarrow$

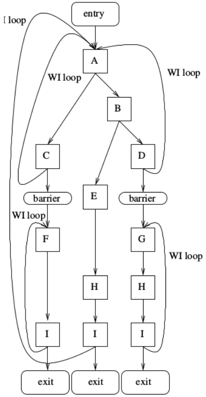


Figure: Tail duplicated WI function[pg. 18]

Figure: Work group function[pg. 18]

Parallel Region formation: Conditional barriers

- Further optimization to the kernel is done by *"loop-peeling"* i.e. by unrolling the first iteration of the loop.
    - We end up with longer branch-less codes ⇒ better ILP optimizations

```
void loop_func(float *a, float*b)
{
    for(int i=0; i<10; i++)
    {
        if(i==0)
            b[i] += a[i];
        b[i] += a[i];
    }
}
```

$\longrightarrow$

```
void peeled_func(float *a, float*b)
{
    /*First iteration*/
    b[0] += a[0];
    b[0] += a[0];

    for(int i=1; i<10; i++)
    {
        b[i] += a[i];
    }
}
```

Parallel Region formation: Conditional barriers

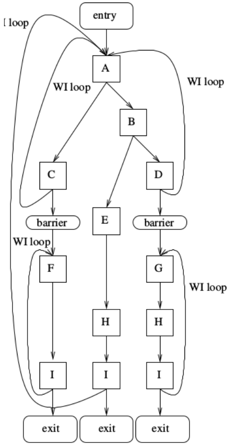


Figure: Work group function[pg. 18]

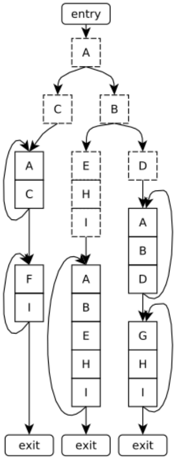
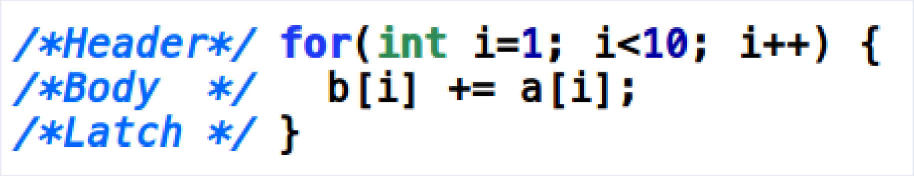


Figure: Peeled work group function[pg.19]

Barriers in kernel loops

Sections of a loop


```
/*Header*/ for(int i=1; i<10; i++) {
/*Body  */    b[i] += a[i];
/*Latch */ }
```

Algorithm

- All work items must execute the b-loop in lock steps
- Add an implicit barrier after the *header*, and before the latch
- The parallel regions lies in between the barriers

Barrier in kernel loops

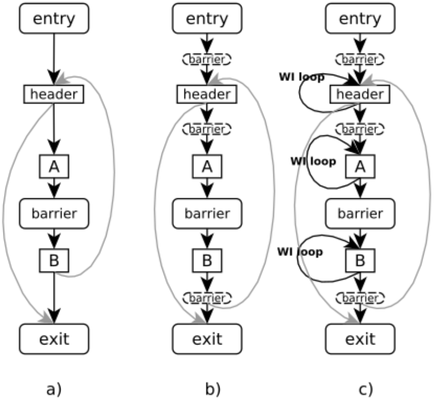


Figure: Parallel regions in *b-loops*[pg.22]

Horizontal Inner loop parallelization

- WI functions with loops having dependency patterns which are difficult to parallelize can be optimized using the work-group functions
- `pocl` strategy: introduce implicit barriers so that the scheduling of the WI captures the parallelism within the WG functions captures the parallel structure of computations

Horizontal Inner loop parallelization example

```
__kernel void horiz_parallel(__global float* a,
                             int blockWidth)
{

    /*Parallel_WI_loop*/
    int gid = get_global_id(0);
    float acc = 0.0;
    for(int i=0; i < blockWidth; ++i)
    {
        acc += (a[blockWidth + i]);
    }
    // ...(more computations)
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

$\longrightarrow$

```
__kernel void horiz_parallel(__global float* a,
                             int blockWidth)
{

    int gid = get_global_id(0);
    float acc = 0.0;
    for(int i=0; i < blockWidth; ++i)
    {
        /*Parallel_WI_loop*/
        acc += (a[blockWidth + i]);
        //[pocl:] adds an implicit barrier here
    }
    // ...(more computations)
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

Handling private variables

Possible solution

A Context Array for each private variable.

- Maintaining an array for each private variable would be inefficient, as it is possible that a private variable is only allocated for a single WI

Implemented heuristic:

- Analyze the LLVM IR
  - If more than 1 WI allocate a private variable then allocate a context array for the private
  - Else, allocate such private variables(allocated by only 1 WI) in one of the registers
- More optimizations are done by doing a uniformity analysis of private variables

Vectorized Mathematical Library Functions

- `pocl` also contains low level implementations of mathematical functions on vector data types though the `VecMathlib` library
- `VecMathlib` is implemented for several backends – `Intel`, `AMD`, `GPUs`
- Backend generic implementation through data structures such as `realvec<double, 2>`
- Implementation Strategy:
    - Try to implement the vector data types and map the vectorized operations to corresponding instructions in the hardware
    - If hardware does not support the instruction then, fallback to serial execution

Implementation details of `VecMathlib`

- IEEE floating point system
- Computations of some mathematical functions through iterative scheme. For example: $\sqrt{x}$, $\frac{1}{x}$
- Trigonometric mathematical functions are reduced to the range $[0, \pi/2]$ and then computed using interpolation through Chebyshev polynomials

Performance Evaluation

Testing platform I:

- Intel x86-64 i7-440
- OpenCL implementations for comparison
  - AMD APP SDK
  - Intel's OpenCL implementation

Testing platform II:

- ARM Cortex-A9
- Neon SIMD Units
- OpenCL implementations for comparison: `FreeOCL`

Testing platform III:

- Cell BE of PS3
- AltiVec instruction set of accessing SIMD units
- OpenCL implementations for comparison: `IBM OpenCL`

Performance comparison on Intel i7

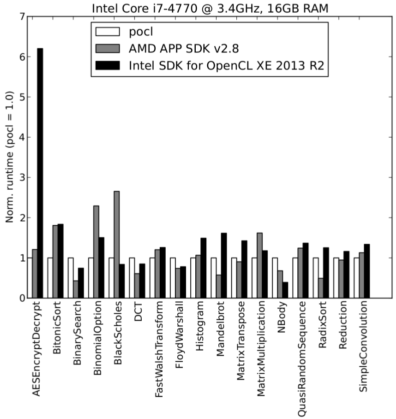


Figure: Comparison of `OpenCL` implementations on Intel i7(*lower is better*)[pg.30]

Performance comparison on AMD Cortex-A9

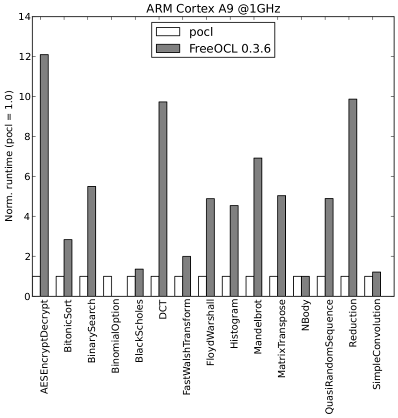


Figure: Comparison of `OpenCL` implementations on AMD Cortex-A9(*lower is better*)[pg.29]

Performance comparison on Cell BE

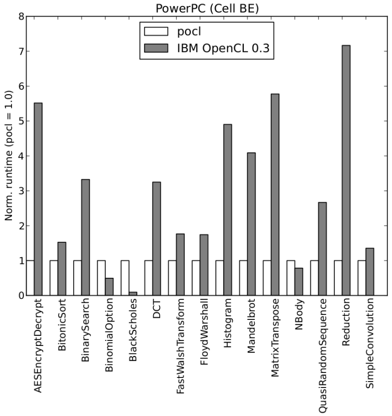


Figure: Comparison of `OpenCL` implementations on Cell BE(*lower is better*)[pg.31]

Current state of pocl

- Support to NVIDIA CUDA is added though LLVM NVPTX backend

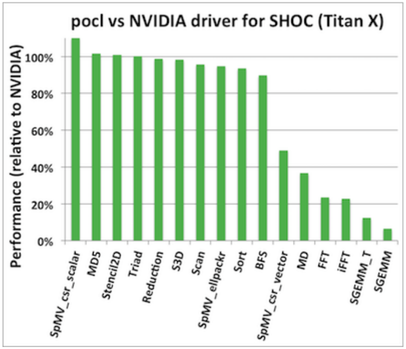


Figure: pocl's OpenCL implementation for NVIDIA[2]

[2]http://portablecl.org/cuda-backend.html

Summary

- The central theme of the work is around the concept of creating/exposing parallel regions of the kernel using compiler transformations, and annotate the embarrassingly parallel sections of code in LLVM IR
- Proofs for the validity of the compiler transformation have been provided in the paper
- Implemented `VecMathlib`, a backend generic library for mathematical functions that evaluates mathematical functions for vector data types
- The authors have shown comparable performance with the vendor's provided OpenCL implementation
- Support for more targets is being added: `NVIDIA` and `pHSA` being the major ones