ADAPTIVE OPENCL LIBRARIES FOR PLATFORM PORTABILITY

Paul Fox
Why OpenCL?

- OpenCL is the most widely-supported massively-parallel programming environment:
  - AMD, NVIDIA, and Intel GPUs.
  - Intel and AMD for CPUs.
  - Intel Xeon Phi.
  - Mobile GPUs from Qualcomm & ARM.
  - Altera and Xilinx FPGAs

These don’t all implement the standard in quite the same way, however.
PORTABILITY CHALLENGES

» Many OpenCL libraries are not even functionally portable.
  » AMD’s clMath libs don’t work well on non-AMD hardware.
  » A lot of testing is required to ensure portability.

» Take into account differences in implementation details: not all vendors implement the standard the same way.

» Very limited performance portability: tuning for one device might make code much slower on another!
Many useful programs can be broken down into a set of primitive operations that have inherent parallelism.

- BLAS Algorithms for numerical computing
- FFT, FIR, etc for image and signal processing.
- Other primitives such as prefix sum, adjacent difference, radix sort, etc.

Provide a set of common algorithms that can be quickly tuned for widely varying hardware, and many programs can be made performance-portable with much less effort required.
TOWARDS PERFORMANCE-PORTABLE KERNELS
KERNELS AND PARALLELISM

» Each work-item executes the kernel code concurrently.

» For best performance, kernels must ensure two constraints are met:
   1. There are enough work-items to fully utilize the hardware.
   2. Each work-item has enough work to make up for the overhead of scheduling it.

» SIMD-style parallelism can be exploited within a work-item through the use of explicit vectors.
Kernels and Control Flow

» Kernels are ideal for performing bulk data processing.
» Avoid complex control flow in kernels.
» Avoid writing a kernel that handles multiple cases.
» Define as much as possible statically.
» Presence of SIMD-style execution.
» Degree of parallelism
» Memory hierarchy layout
» Preferred integer operation width
» Cost of a work-group barrier
» Cost of branching
<table>
<thead>
<tr>
<th>Parameter</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD-style Execution</td>
<td>Yes</td>
<td>Sometimes</td>
</tr>
<tr>
<td>Degree of Parallelism</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Memory Layout</td>
<td>Shallow (Cached main memory, few registers)</td>
<td>Deep (Possibly cached main memory, local scratchpad memory, many registers, special memory)</td>
</tr>
<tr>
<td>Preferred integer operation width</td>
<td>32-bit</td>
<td>24-bit (can use FPU) or 32-bit (with dedicated integer arithmetic unit)</td>
</tr>
<tr>
<td>Cost of Branching</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Cost of Barrier</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>
WRITING A CPU KERNEL

» Avoid register spilling (few private variables)
» Implicit locality (no local memory)
» Avoid duplicating work between work-items (low parallelism, low branch cost)
» Avoid using work-group barriers (high barrier cost)
» Use 32-bit integer arithmetic (avoid 64-bit pointer arithmetic).
» Make use of vector types for more fine-grained control (SIMD)
» Bottom Line: simple, straightforward kernel.
Writing a GPU Kernel

- May not benefit from vector types
- Use local memory and registers whenever possible (deep memory layout, many registers).
- Avoid branching and synchronization, even if it means duplicating work.
- Unroll inner loops as much as possible (preferably fully)
- Use 32-bit or 24-bit integer arithmetic where possible, depending on device preference.
- Bottom Line: static unrolling, manual caching, and indexing to ensure in-order memory accesses become crucial to performance.
A Tale Of Two Kernels

CPU-Optimized

unsigned int row = get_global_id(0);
const __global float* arow = &a[row*lda];

float ytemp = 0.f;
prefetch(arow, BLOCK_SIZE);

uint i=0; i < n; i += BLOCK_SIZE) {
    prefetch(arow+BLOCK_SIZE, BLOCK_SIZE);
    #pragma unroll BLOCK_SIZE
    for(unsigned int ii=0; ii < BLOCK_SIZE; ++ii)
        ytemp += arow[ii] * x[i+ii];
arow = &arow[BLOCK_SIZE];
}

for(unsigned int ii=0; ii < n; ++i)
    ytemp += arow[ii] * x[i+ii];
y[row] = ytemp;

Haswell i7-4770
CPU-Optimized:  5955 µs
GPU-Optimized: 12118 µs

NVIDIA GTX 460
CPU-Optimized: 38315 µs
GPU-Optimized: 4212 µs

GPU-Optimized

float ytemp = 0.f;
__local float ablock[BLOCK_SIZE][BLOCK_SIZE];
__local float xblock[BLOCK_SIZE];

uint globalx = get_global_id(0);
uint localx = get_local_id(0);
uint startrow = get_group_id(0)*BLOCK_SIZE;

if(i=0)
    uint tidx = localx;
    __local float* abrow = &ablock[localx][0];

for(i; i < n; i += BLOCK_SIZE; tidx += BLOCK_SIZE, i += BLOCK_SIZE){
    // Load block into shared memory cooperatively
    xblock[localx] = x[tidx];
    #pragma unroll BLOCK_SIZE
    for(uint ij=0, iij=startrow; ij < BLOCK_SIZE; ++ij, ++iij)
        ablock[ij][localx] = (ijg<m) ? a[ijj*lda+tidx] : 0.f;
    barrier(CLK_LOCAL_MEM_FENCE);
    // Do multiplication of row section
    #pragma unroll BLOCK_SIZE
    for(uint ii=0; ii < BLOCK_SIZE; ++ii)
        ytemp += abrow[ii] * xblock[ii];
    barrier(CLK_LOCAL_MEM_FENCE);
}

if(tidx < n){
xblock[localx] = x[tidx];
#pragma unroll BLOCK_SIZE
for(uint ij=0, iij=startrow; ij < BLOCK_SIZE; ++ij, ++iij)
    ablock[ij][localx] = (ijg<m) ? a[ijj*lda+tidx] : 0.f;
}

barrier(CLK_LOCAL_MEM_FENCE);
if(globalx < m){
    for(uint ii=0; ii < n; ++ii)
        ytemp += abrow[ii] * xblock[ii];
y[get_global_id(0)] = ytemp;
}

Copyright © 2014 EM Photonics – EM Photonics Proprietary
A Tale Of Two Kernels

CPU-Optimized

```c
unsigned int row = get_global_id(0);
const __global float* arow = &a[row*lda];

float ytemp = 0.f;
prefetch(arow, BLOCK_SIZE);
uint i;
for(i=0; i < n-BLOCK_SIZE; i += BLOCK_SIZE) {
    prefetch(arow+BLOCK_SIZE, BLOCK_SIZE);
    #pragma unroll BLOCK_SIZE
    for(unsigned int ii=0; ii < BLOCK_SIZE; ++ii)
        ytemp += arow[ii] * x[i+ii];
arow = &arow[BLOCK_SIZE];
}
for(unsigned int ii=0; ii < n-i; ++ii)
    ytemp += arow[ii] * x[i+ii];
y[row] = ytemp;
```

GPU-Optimized

```c
float ytemp = 0.f;
__local float ablock[BLOCK_SIZE][BLOCK_SIZE];
__local float xblock[BLOCK_SIZE];
uint globalx = get_global_id(0);
uint localx = get_local_id(0);
uint startrow = get_group_id(0)*BLOCK_SIZE;
uint i=0;
uint tidx = localx;
__local float* abrow = &ablock[localx][0];

for(i; i < n-BLOCK_SIZE; tidx += BLOCK_SIZE, i += BLOCK_SIZE){
    // Load block into shared memory cooperatively
    xblock[localx] = x[tidex];
    #pragma unroll BLOCK_SIZE
    for(uint ij=0, ijj=startrow; ij < BLOCK_SIZE; ++ij, ++ijj)
        ablock[ij][localx] = (ijj<m) ? a[ijj*lda+tidx] : 0.f;
    barrier(CLK_LOCAL_MEM_FENCE);
    // Do multiplication of row section
    #pragma unroll BLOCK_SIZE
    for(uint ii=0; ii < BLOCK_SIZE; ++ii)
        ytemp += abrow[ii] * xblock[ii];
    barrier(CLK_LOCAL_MEM_FENCE);
}
if(tidx < n){
    xblock[localx] = x[tidex];
    #pragma unroll BLOCK_SIZE
    for(uint ij=0, ijj=startrow; ij < BLOCK_SIZE; ++ij, ++ijj)
        ablock[ij][localx] = (ijj<m) ? a[ijj*lda+tidx] : 0.f;
}
 barrier(CLK_LOCAL_MEM_FENCE);
if(globalx < m){
    for(uint ii=0; ii < n-i; ++ii)
        ytemp += abrow[ii] * xblock[ii];
y[get_global_id(0)] = ytemp;
}
```

Structurally Similar

• Block prefetch
A Tale Of Two Kernels

CPU-Optimized

unsigned int row = get_global_id(0);
const __global float* arow = &a[row*lda];

float ytemp = 0.f;
prefetch(arow, BLOCK_SIZE);
uint i;
for(i=0; i < n-BLOCK_SIZE; i += BLOCK_SIZE) {
    // Load block into shared memory cooperatively
    xblock[localx] = x[tidx];
    #pragma unroll BLOCK_SIZE
    for(unsigned int ii=0; ii < BLOCK_SIZE; ++ii)
        ytemp += arow[ii] * x[i+ii];
    arow = &arow[BLOCK_SIZE];
}
for(unsigned int ii=0; ii < n-i; ++ii)
    ytemp += arow[ii] * x[i+ii];
y[row] = ytemp;

GPU-Optimized

float ytemp = 0.f;
__local float ablock[BLOCK_SIZE][BLOCK_SIZE];
__local float xblock[BLOCK_SIZE];
uint globalx = get_global_id(0);
uint localx = get_local_id(0);
uint startrow = get_group_id(0)*BLOCK_SIZE;
uint i=0;
uint tidx = localx;
__local float* abrow = &ablock[localx][0];
for(i; i < n-BLOCK_SIZE; tidx += BLOCK_SIZE, i += BLOCK_SIZE) {
    // Load block into shared memory cooperatively
    xblock[localx] = x[tidx];
    #pragma unroll BLOCK_SIZE
    for(uint ij=0, ijj=startrow; ij < BLOCK_SIZE; ++ij, ++ijj)
        ablock[ij][localx] = (ijj<m) ? a[ijj*lda+tidx] : 0.f;
bARRIER(CLK_LOCAL_MEM_FENCE);
    // Do multiplication of row section
    #pragma unroll BLOCK_SIZE
    for(uint ii=0; ii < BLOCK_SIZE; ++ii)
        ytemp += abrow[ii] * xblock[ii];
bARRIER(CLK_LOCAL_MEM_FENCE);
}
if(tidx < n){
    xblock[localx] = x[tidx];
    #pragma unroll BLOCK_SIZE
    for(uint ij=0, ijj=startrow; ij < BLOCK_SIZE; ++ij, ++ijj)
        ablock[ij][localx] = (ijj<m) ? a[ijj*lda+tidx] : 0.f;
}
bARRIER(CLK_LOCAL_MEM_FENCE);
if(globalx < m){
    for(uint ii=0; ii < n-i; ++ii)
        ytemp += abrow[ii] * xblock[ii];
y[get_global_id(0)] = ytemp;
}

Structurally Similar

• Block prefetch
• Unrolled block multiply
unsigned int row = get_global_id(0);
const __global float* arow = &a[row*lda];

float ytemp = 0.f;
prefetch(arow, BLOCK_SIZE);
uint i;
for(i=0; i < n-BLOCK_SIZE; i += BLOCK_SIZE) {
prefetch(arow+BLOCK_SIZE, BLOCK_SIZE);
#pragma unroll BLOCK_SIZE
for(unsigned int ii=0; ii < BLOCK_SIZE; ++ii)
  ytemp += arow[ii] * x[i+ii];
arow = &arow[BLOCK_SIZE];
}
for(unsigned int ii=0; ii < n-i; ++ii)
  ytemp += arow[ii] * x[i+ii];
y[row] = ytemp;

CPU-Optimized

Structurally Similar
• Block prefetch
• Unrolled block multiply
• Unblocked end computation

GPU-Optimized

float ytemp = 0.f;
__local float ablock[BLOCK_SIZE][BLOCK_SIZE];
__local float xblock[BLOCK_SIZE];
uint globalx = get_global_id(0);
uint localx = get_local_id(0);
uint startrow = get_group_id(0)*BLOCK_SIZE;
uint i=0;
uint tidx = localx;
__local float* abrow = &ablock[localx][i];

for(; i < n-BLOCK_SIZE; tidx += BLOCK_SIZE, i += BLOCK_SIZE){
  // Load block into shared memory cooperatively
  xblock[localx] = x[tidx];
  #pragma unroll BLOCK_SIZE
  for(uint ij=0, ijg=startrow; ij < BLOCK_SIZE; ++ij, ++ijg)
    ablock[ij][localx] = (ijg<m) ? a[ijg*lda+tidx] : 0.f;
  barrier(CLK_LOCAL_MEM_FENCE);
  // Do multiplication of row section
  #pragma unroll BLOCK_SIZE
  for(uint ii=0; ii < BLOCK_SIZE; ++ii)
    ytemp += abrow[ii] * xblock[ii];
  barrier(CLK_LOCAL_MEM_FENCE);
}
if(tidx < n){
  xblock[localx] = x[tidx];
  #pragma unroll BLOCK_SIZE
  for(uint ij=0, ijg=startrow; ij < BLOCK_SIZE; ++ij, ++ijg)
    ablock[ij][localx] = (ijg<m) ? a[ijg*lda+tidx] : 0.f;
  barrier(CLK_LOCAL_MEM_FENCE);
}
if(globalx < m){
  for(uint ii=0; ii < n-i; ++ii)
    ytemp += abrow[ii] * xblock[ii];
y[get_global_id(0)] = ytemp;
Towards a Portable Kernel

» How do we combine different optimization approaches?

» Need to be able to factor code in or out depending on target device.

» Build parameterized “stencil” kernel with basic structure of algorithm.

» Supply different parameters to stencil based on properties of target platform.
BUILDING A STENCIL

» C macros: supply parameters to kernel compiler.
   » Minimal extra code required.
   » Can become difficult to develop/debug as kernel complexity and parameter space increase.

» C++ templates: only work on AMD at present.

» **Code generation:** use tools to generate kernel code based on generic algorithm specification.
   » Requires more work to set up.
   » Doesn’t depend on the OpenCL compiler in any way.
   » Sweet spot is somewhere between a template system and a full language.
Typically, the programmer will not know what combination of possible kernel generation parameters will yield the best performance. Ideally, this can be determined in an automated fashion, by using an algorithm to autotune kernels. Finding an optimal solution is probably intractable in most cases, but we can usually find a solution that exhibits relatively high performance.
AUTOTUNING MATRIX MULTIPLICATION

» Started with a kernel stencil and an exhaustive search.
» Good results, but exhaustive search is very time consuming.
» Few people want to spend hours or days running tuning passes.
Two Approaches to Fast Tuning

» Heuristics based on device information
  » OpenCL allows querying devices for certain parameters.
  » These can serve as a guide for an initial guess which may exhibit acceptable performance.

» Device database
  » A database of known good tuning parameters for different devices.
  » Similar devices can use these as starting points for tuning (e.g. a GTX780 and a GTX770 will likely have similar optimal parameters).

» AI techniques can be used based on explicit and implicit “device models”
  » Some work has been done in the past, and we intend to investigate this further in the future.
CODE GENERATION AND REUSABILITY
Kernels are typically presented to the OpenCL API as strings, and compiled by the implementation.

Support for OpenCL 1.2 and SPIR are still not universal, so we can’t build libraries or assemble our own IR ahead of time.

How do we reuse code, then?
Generating kernels on-the-fly lets us do more than tune for performance.

We can build snippets of code that can be individually tuned and reused.

We are currently working on this for BLAS

SYRK (Symmetric Rank-K Update) can be implemented in terms of sub-operations consisting of block-level SYRK and GEMM (matrix-matrix multiply) algorithms.

This way, users can request code following a particular interface to insert into their own kernels.
AFFINIBLAS

We are currently working on a fully-portable OpenCL BLAS implementation to be released as open source software.

Testing on NVIDIA and AMD GPUs, Intel and AMD CPUs, and Xeon Phi.

We plan to also test on mobile platforms and hopefully FPGAs.
We will be working LAPACK and FFT libraries after BLAS is fully functional.

Our first priority is portability: these libraries should be able to function on any hardware for which an OpenCL implementation exists.

Offer the tools to tune for specific hardware and plug-in optimized implementations of various pieces.

kernels as well as block-level algorithms for inserting into user kernels will be available.
CALL FOR CONTRIBUTORS

» Once we make our initial release, we will need contributors to keep development going.
   » We are a small team, and cannot test and tune the algorithms for all possible hardware.
   » Community involvement will be key to getting continuing to improve the code.

» We hope that many people find our code useful enough to make contributions!
Mailing List

» To get release announcements, subscribe to our mailing list by mailing to:

affinimath+subscribe@emphotonics.com

Or join via Google Groups (affinimath)

» Technical questions may be sent to Paul Fox (fox@emphotonics.com)