OpenCL

Jiří Filipovič

fall 2017

OpenCL

What is OpenCL?

- an open standard for heterogeneous systems programming
- low-level, derived from C, HW abstraction very similar to CUDA

Advantages over CUDA

- can be used for wide area of HW
- open standard, independent on a single corporation

Disadvantages compared to CUDA

- more complex API (similar to CUDA Driver API)
- often less mature implementation
- slower implementation of new HW features

Portability

One implementation can be compiled for different types of HW

- if we do not use extensions ...

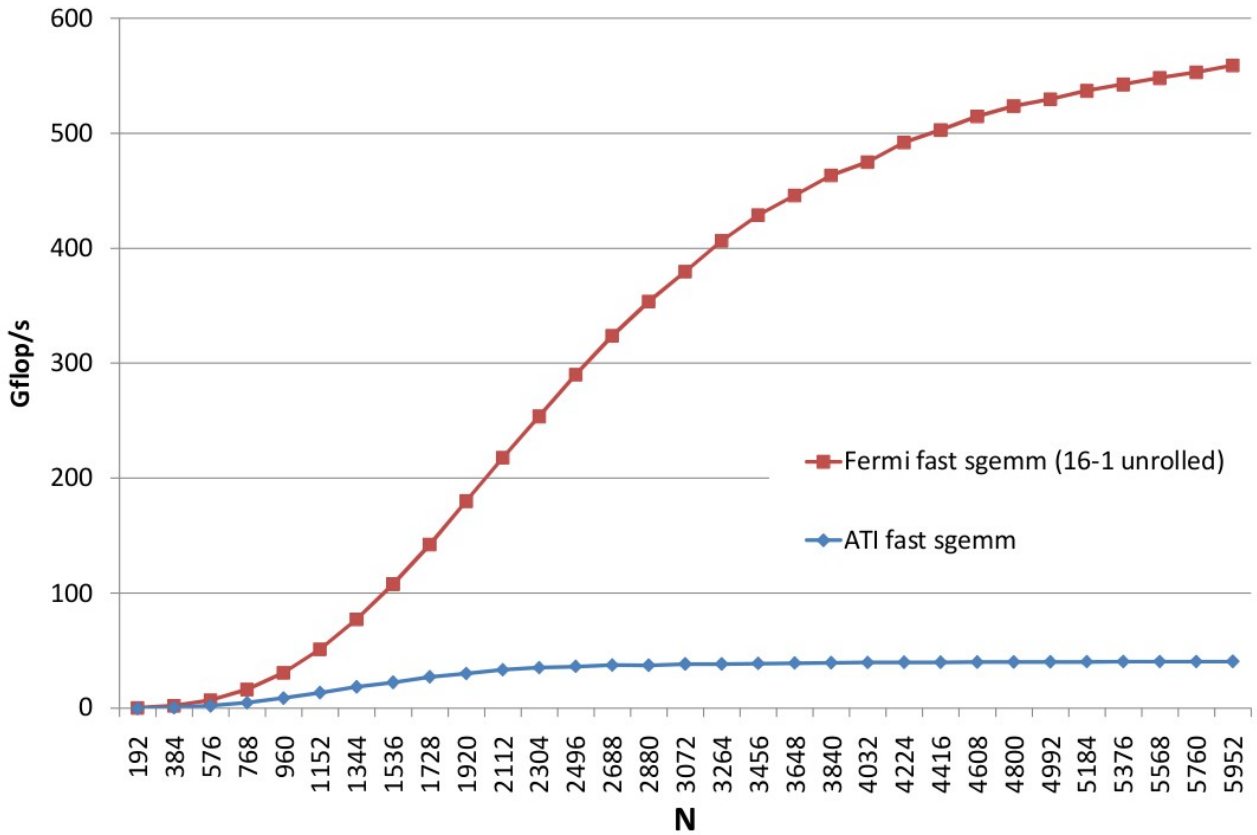
However, the implementation optimized for some type of HW may be very slow on another HW

- we need to re-optimize for different HW architectures

So, it is the standard for programming of various types of HW, but we need to write different kernels for different architectures.

- high importance easily modifiable code or autotuning

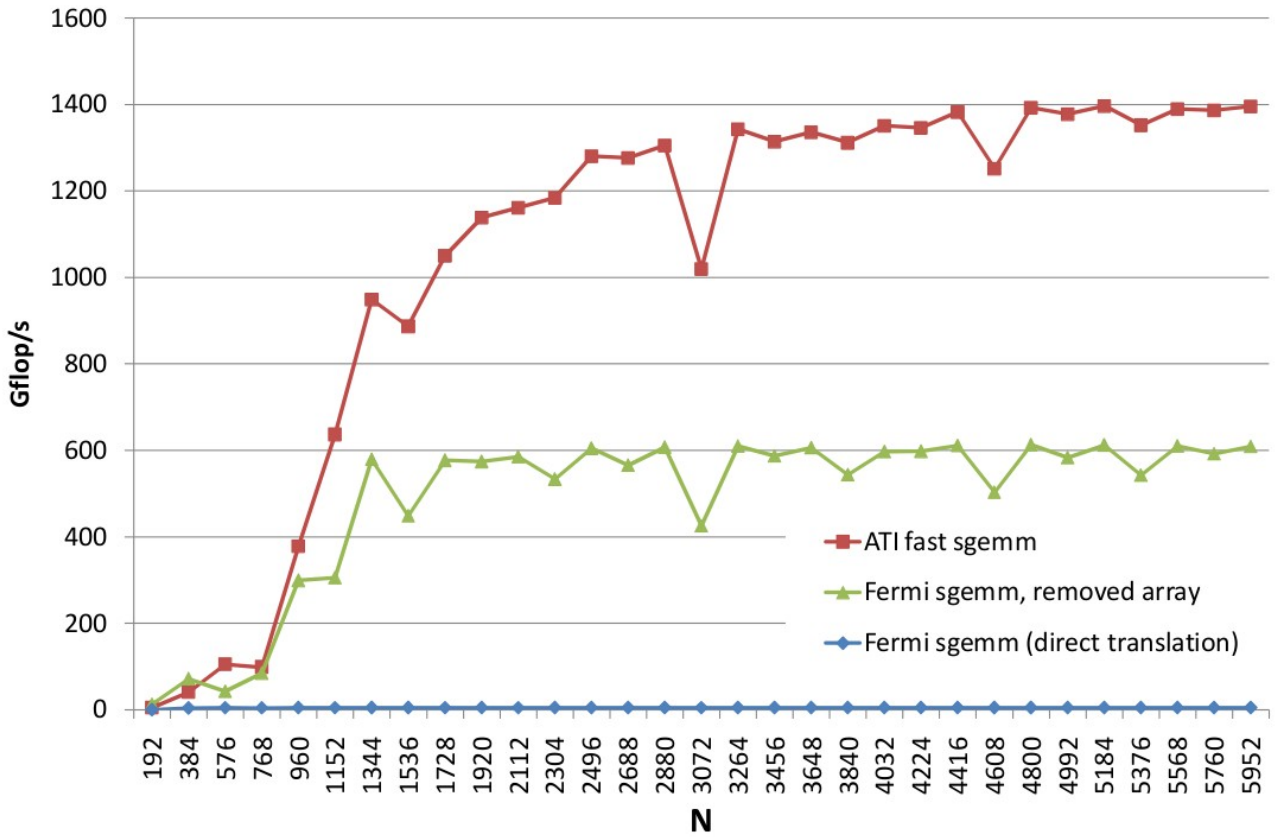
Performance Portability



Obrázek : SGEMM optimized for Fermi and Cypress, running on Fermi[1].

[1] Du et al. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming

Performance Portability



Obrázek : SGEMM optimized for Fermi a Cypress, running on Cypress[2].

[2] Du et al. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming

Main Differences

OpenCL is not integrated to C/C++

- the OpenCL kernel is stored as a string, which is usually compiled during program execution
- kernel cannot share code with C/C++ codebase (user-defined types, common functions etc.)

Kernels in OpenCL do not use pointers

- we cannot dereference, use pointer arithmetics, link different buffers
- we can traverse the buffer by index, of course

OpenCL is strictly derived from C

- no C++ stuff

OpenCL uses queues for HW devices

- eases using multiple devices/streams

Queues can work out-of-order

- eases load balancing

CUDA-OpenCL dictionary

Main differences in terminology

CUDA	OpenCL
multiprocessor	compute unit
scalar processor	processing element
thread	work-item
thread block	work-group
grid	NDRange
shared memory	local memory
registers	private memory

Vector Addition – Kernel

CUDA

```
__global__ void addvec(float *a, float *b, float *c)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

OpenCL

```
__kernel void vecadd(__global float * a, __global float * b,
__global float * c)
{
  int i = get_global_id(0);
  c[i] = a[i] + b[i];
}
```

Vector Addition – Host Code

To execute the kernel, we need

- to define a platform
  - device (at least one)
  - context
  - queues
- allocate and copy data
- compile the kernel code
- configure the kernel and execute it

Vector Addition – Platform Definition

```
cl_uint num_devices_returned;
cl_device_id cdDevice;
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1,
&cdDevice, &num_devices_returned);

cl_context hContext;
hContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &err);

cl_command_queue hQueue;
hQueue = clCreateCommandQueue(hContext, hDevice, 0, &err);
```

Vector Addition – Platform Definition

The platform can have more devices

- can be selected by the type (e.g. a GPU)
- can be selected by vendor
- we can also choose HW using finer informations
  - number of cores
  - frequency
  - memory size
  - extensions (double precision, atomic operations etc.)

Each device needs at least one queue

- cannot be used otherwise

Vector Addition – Memory Allocation and Copy

```
cl_mem hdA, hdB, hdC;
hdA = clCreateBuffer(hContext, CL_MEM_READ_ONLY,
  cnDimension * sizeof(cl_float), pA, 0);
...
```

There is no explicit copy – allocation and copy is performed in lazy fashion, i.e. in time when data are needed. Consequently, the target device is not defined in the memory allocation.

Vector Addition – Kernel Execution

```
const unsigned int cnBlockSize = 512;
const unsigned int cnBlocks = 3;
const unsigned int cnDimension = cnBlocks * cnBlockSize;

cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1, sProgramSource
  ,0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0);

cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "addvec", 0);

clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hdA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hdB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hdC);

clEnqueueNDRangeKernel(hQueue, hKernel, 1, 0, &cnDimension
  , &cnBlockSize, 0, 0, 0);
```

Vector Addition – Cleanup

```
clReleaseKernel(hKernel);
clReleaseProgram(hProgram);
clReleaseMemObj(hdA);
clReleaseMemObj(hdB);
clReleaseMemObj(hdC);
clReleaseCommandQueue(hQueue);
clReleaseContext(hContext);
```

AMD VLIW GPU Architecture

Older processors

- Evergreen and Northern Islands

We will discuss main differences between AMD and NVIDIA GPU

- the rest is very similar

Main differences

- VLIW architecture
- two memory access modes – the fast path and complete path
- less sensitive to misaligned access, more sensitive to partition camping analogy
- wavefront (the warp analogy) has 64 threads

VLIW Architecture

VLIW

- the instruction word includes several independent operations
- static planning of instruction parallelism (dependencies analyzed during compilation)
- allows higher density of ALUs
- threads should perform a code with sufficient instruction parallelism and a compiler needs to recognize it
  - easier in typical graphics tasks than general computating ones
- AMD GPU implements VLIW-5 or VLIW-4, 1 instruction is SFU

Optimizations for VLIW

Explicit vectorization

- we work with vector variables (e.g. float4)
- generation of VLIW is straightforward for the compiler

Implicit generation of VLIW

- we write a scalar code
- compiler tries to recognize independent instruction and create VLIW code
- we can help the compiler by unrolling and grouping the same operations performing different iterations

Optimizations for VLIW

Issues with VLIW

- higher consumption of on-chip resources per thread (unrolling, vector types)
- we need independent instructions
  - problematic e.g. with conditions
- together with large wavefront it is highly sensitive to divergence

Global Memory Access

Fast path vs. complete path

- fast path is significantly faster
- fast path is used for load/store of 32-bit values
- complete path is used for everything other (values of different size, atomics)
- the compiler needs to explicitly use one of those paths
  - access path is the same for the whole buffer, so we can degrade the global memory bandwidth easily

Fast path vs. complete path

```
__kernel void
CopyComplete(__global const float * input, __global float* output)
{
  int gid = get_global_id(0);
  if (gid < 0){
    atom_add((__global int *) output,1);
  }
  output[gid] = input[gid];
  return ;
}
```

Difference on Radeon HD 5870: 96 GB/s vs. 18 GB/s.

Global Memory Access

Permutation of thread-element mapping in wavefront

- small penalization ($< 10\,\%$)
- better than c.c. < 1.2

Faster access using 128-bit in single instruction

- e.g. accessing float4
- 122 GB/s instead 96 GB/s using HD 5870 and the memory copy example

Memory Channels

Radeons of 5000 series have memory channels interleaved by 256 bytes

- all threads within wavefront should use the same channel
  - wavefront accessing the aligned contiguous block of 32-bit elements (with arbitrary permutation of thread-element mapping) uses the same channel
- if multiple channels are accessed by wavefront, the access is serialized
  - occurs e.g. in misaligned access

Bank and Channel Conflicts

Analogy of partition camping

- the global memory is accessed using banks and channels
- concurrent workgroups should access via different channels and different banks
  - bandwidth is limited otherwise
- the arrangement of banks depends on the number of channels
  - for instance, 8 channels means that the bank switches every 2 KB
- high penalization of accessing the same channel and the same bank (0.3 vs. 93 GB/s on Radeon HD 5870)

Local Data Storage

Local Data Storage (LDS) is very similar to NVIDIA's shared memory

- composed of 32 or 16 banks
- the quarter-wafefront needs to access different banks simultaneously
  - otherwise the bank conflicts appear
  - in the case of 32 banks we can efficiently use float2
- broadcast is supported for a single value (analogy of c.c. 1.x)

AMD GCN GPU Architecture

Nowadays architecture, known as Graphic Core Next.
Significantly different than previous generations

- no VLIW, compute unit contains one scalar processor and four vector processors
  - the code performed by threads is scalar (vectorized code usually slower because of resource consumption)
  - conditions penalization is lower compared to VLIW
- L1 cache for read and write
- concurrent kernel invocations