ECE 574 – Cluster Computing
Lecture 17

Vince Weaver

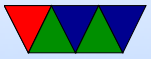
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

6 April 2017

Announcements

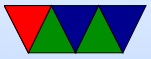
- HW#8 will be posted
- HW#7
  Power outage
  Pi Cluster
  Runaway jobs (tried setting limit lower, but the default sbatch I gave you times out at 2 hours :( )



CUDA Examples

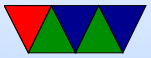
- Make builds them. .cu file, built with nvcc
- `./hello_world` bit of a silly example
- `saxpy.c`
  single a*x+y

  CPU GPU run 320000000 1.12s 2.06
- What happen if thread count too high? Max threads per block 512 on Compute 2, 1024 on compute 3 Above



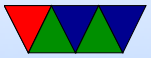
1024? Try saxpy_block

- maximum block size 64k on Compute version 2, 2GB on Compute Version 3 200,000 50,000 cpu = 4.5s gpu = 0.8s



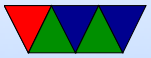
CUDA Tools

- `nvidia-smi`. Various options. Usage, power usage, etc.
- `nvprof ./hello_world` profiling
- `nvvp` visual profiler, can't run over text console
- `nvidia-smi --query-gpu=utilization.gpu,power.dr`
  `--format=csv -lms 100`

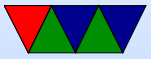


OpenCL

- CUDA is only for NVIDIA
- What if you have Intel or AMD (ATI) chip? Or ARM MALI (sadly not Raspberry Pi)
- OpenCL is sort of like CUDA, but cross-platform
- Not only for GPUs, but can target regular CPU, DSP, FPGAs, etc
- Vendor provides a driver

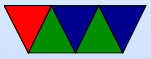


- Khronos (the OpenGL + Vulkan people?) also run OpenCL
- Windows, OSX, Linux



OpenCL

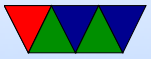
-



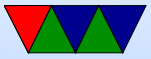
SAXPY

Single-precisison A*X+Y vector add

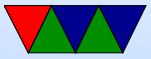
```
void saxpy(int n, float a, float *x, float *y, float *out) {
    for(i=0;i<n;i++) {
        out[i]=a*x[i]+y[i];
    }
}

#include <CL/cl.h>

const char *saxpy_kernel=
"__kernel\n"
"void saxpy_kernel(float a,\n"
"    __global float *A,\n"
"    __global float *B,\n"
"    __global float *out) {\n"
"    int index=get_global_id(0);\n"
"    out[index]=alpha*A[index]+B[index];\n"
"}\n";

#define N 1024
```



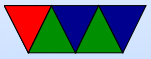
```
int main(int argc, char **argv) {
    int i;
    float alpha=5.0;
    float *A=(float *)malloc(sizeof(float)*N);
    float *B=(float *)malloc(sizeof(float)*N);
    float *C=(float *)malloc(sizeof(float)*N);

    for(i=0;i<N;i++) { A[i]=i; B[i]=10*i; C[i]=0;}

// get platform
cl_platform_id *platforms=NULL;
cl_uint num_platforms;
cl_int clStatus=clGetPlatformIDs(0,NU,&num_platforms);
platforms=(cl_platform_id 8)malloc(sizeof(cl_platform_id)*num_platforms);
clStatus=clGetPlatformIDs(num_platforms, platforms,NULL);

cl_device_id *device_list=NULL;
cl_uint num_devices;
clStatus = clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_GPU,0,NULL,&num_devices);
device_list=(cl_device_id *)malloc(sizeof(cl_device_ids)*num_devices);
clStatus=clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_GPU,num_devices,device_list,NULL
// create context
cl_context context;
```



```
context=clCreateContext(NULL, num_devices,device_list,NUL,NULL,&clStatus);
// Create command queue
cl_commad_queue command_queue=clCreateCommandQeueu(contet,device_list[],0,&clStatus)
// create memory buffer on device
cl_mem A_clmem=clCreateBuffer(context,CL_MEM_READ_ONLY,VECETOR_SIZE*sizeof(float),NU
cl_mem B_clmem=clCreateBuffer(context,CL_MEM_READ_ONLY,VECETOR_SIZE*sizeof(float),NU
CL_MEM_WRITE_ONLY cl_mem C_clmem=clCreateBuffer(context,CL_MEM_READ_ONLY,VECETOR_SIZ

// Copy buffer a and b to device
clStatus=clEnqueueWriteBuffer(command_Queue, A_clmem, CL_TRUE,0,VECTOR_SIZE*sizeof(f
clStatus=clEnqueueWriteBuffer(command_Queue, A_clmem, CL_TRUE,0,VECTOR_SIZE*sizeof(f

// create a program
cl_program program=clCreatePrograWithSource(context,1,(const char **)&saxpy_kernel,N
// build program
clStatus=clBuildProgram(program,1,device_list,NULL,NULL,NUL);
// create OpenCL kernel
cl_kernel kerne;=clCreateKernel(program,"saxmpu_kernel",&clStatus);

// Set Arguments
clStatus=clSetKernelArg(kernel,0,sizeof(float),)coid *)&alpha);
clStatus=clSetKernelArg(kernel,1,sizeof(cl_mem),(coid *)&A_clmem);
clStatus=clSetKernelArg(kernel,2,sizeof(cl_mem),
(void *)&B_clmem);
```



```
CclStatus=clSetKernelArg(kernel,2,sizeof(cl_mem),
(void *)&B_clmem);

// excute kernel

size_t global_size=VECTOR_SIZE;
size_t local_size=64;
clStatus=clEnqueueNDRangeKernel(command_Queue,ekernel,1,NULL,&global_szize,&loca-siz

// Read out results
clStatus=clEnqueueReadBuffer(command_queue,C_clemem, CL_TRUE,0,VECTOR_SIZE*sizef(flo

// cleanp
clStatus=clFlush(command_queue);
clStatus=clFinish(command_queue);

// display results
for(i=0;i<VECTOR_SIZE;i++)
printf("%d %d %d %d alhpha"A[i],B[i],Cpi[);
// release
clstatus=ClReleaseKenrle(kernel);
clReleaseProgram(program);
clReleaseMemoObject(A_clmem);
clstatus=clReleaseCommandQueue(command_Queue);
```

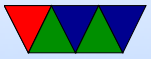


```
clStatus=clReleaseContet(context);
free(A);BCplatformsdevice_list
```

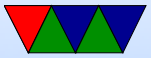
gcc -I include -L /lib -lOpenCL saxpyc -o saxxpy

Open CL program Flow

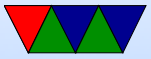
- Allocate host buffer
- Get platform/device
- Set up platform
- Choose device



- Create context
- Create command queue
- Create memory buffer on device
- Copy buffer to device
- Create a program kernel
- Build kernel
- Set arguments

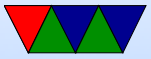


- Execute
- Read back results
- clean up and wait to finish
- Release



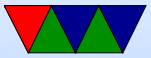
Platforms

- Query number of platforms
- clGetPlatformIDs
- Then malloc space, and use same function to get info
- Can iterate and get NAME, VENDOR, VERSION



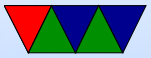
Devices

- Now when got platform, similarly clGetDeviceIDs
- Why multiple? Intel or AMD CPU might have both CPU and GPU



kernel

- A lot like CUDA, where split into 1D, 2D, or 3D grid.
- get_global_id();
- get_local_id();
- get_num_groups();
- get_group_size()
- get_group_id()



clEnqueueNDRangeKernel(command_Queue kernel, work_dim, global_work_offset, global_work_size, local_work_si event_wait_list event

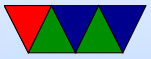
Context

Command Queue FIFO or out of order (always issued in order)

Memory Model? relaxed?

Global/Constant/Local memory Private Memory

OpenCL ICD (installable client driver)



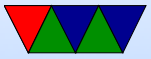
Memory flags. r/w, ro, wo Whether host pointer or not Using host pointers might not be fast if depending on arch sub-buffer?

copying mem host to device device to device

OpenCL images... special operations on image data

Creating Kernels clCreateProgramWithSource() clCreatePr
If compile with source, might need to query the build error logs when something went wrong

Why use binary file? Proprietary Not have to build every time run



SPIR – standard portable Intermediate Representation

Setting kernel arguments must do this before running

executing kernel

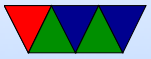
querying kernel

built-in kernels?

Synchronization when neeed? single device, out of order queue multiple devices?

coarse grained clFlush/clFinish

fine grained event based



memory fences?

CL event, for communcating

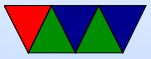
OpenCL C Programming

own build in data types basic app vector app_vector char cl_char charn cl_charn etc

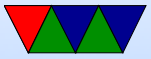
why? portable. sadly sizes not same on windows/linux

n elene,t 2,3,4,8,16 sizes “half” type for 16-bit fp

address space qualifiers __global __local __constant __private



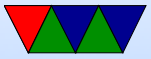
OpenACC?



Other Accelerator Options

- XeonPhi – came out of the larabee design (effort to do a GPU powered by x86 chips). Large array of x86 chips(p5 class on older models, atom on newer) on PCIe card. Sort of like a plug-in mini cluster. Runs Linux, can ssh into the boards over PCIe. Benefit: can use existing x86 programming tools and knowledge.

- FPGA – can have FPGA accelerator. Only worthwhile if you don't plan to reprogram it much as time delay in reprogramming. Also requires special compiler support



(OpenMP?)

- ASIC – can have hard-coded custom hardware for acceleration. Expensive. Found in BitCoin mining?
- DSPs – can be used as accelerators