nag®

CSC / NAG Autumn School on

Core Algorithms in High-Performance Scientific Computing

Libraries V

David Quigley

GPUs for linear algebra

GPUs for Linear Algebra?

D. Quigley

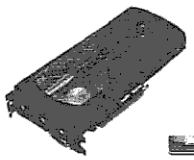
Department of Physics
University of Warwick

Introduction

- Phil's eigensolver code from yesterday's practical is a good model for a real plane-wave electronic structure code.
- It is dominated by two computationally heavy steps.
  - Orthogonalisation (matrix-matrix multiply)
  - Appling potential to wave function (two 3D FFTs)
- Both are candidates for GPU acceleration.
- Ran an UG summer project (2010) to port the practical to the GPU to determine if acceleration of CASTEP might be feasible.

CUDA or OpenCL?

- CUDA is not an open standard and is specific to NVIDIA hardware.
- NVIDIA may choose to break backwards compatibility with older hardware at any point.
- OpenCL is an industry standard for hybrid computing using GPUs and other accelerators (e.g. cell).
- Easy decision – use OpenCL!



A simple CUDA program

```
#include <stdio.h>

float  h_a; // pointer to a in host memory
float  d_a; // pointer to a in device memory


// This is a kernel as indicated by the __global__ qualifier. Global means
// visible to the device *and* the host which invokes it. This kernel
// squares a number in an array.
__global__ void kernel_Sq(float* a) {

  // Built-In variables (can be multidimensional)
  // BlockDim  = how many threads per block?
  // blockIdx  = which block am I in?
  // threadIdx = which thread am I within the block?

  int i = blockDim.x * blockIdx.x + threadIdx.x;

  a[i]  = a[i]*a[i];

}
```



```
// Host code - this is executed by the CPU as normal. Compile with nvcc
int main(int argc, char** argv) {

  int n = 262144;

  // Allocate memory for h_a and populate on the host
  h_a   = (float *)malloc(n*sizeof(float));
  for (i=0;i<n;i++) { h_a[i] = (float)(i); }

  // Allocate memory on device, note cudaMalloc passes pointer by reference
  cudaMalloc(&d_a,n*sizeof(float));

  // Copy a into device memory using a helper function
  cudaMemcpy(d_a,h_a,n*sizeof(float),cudaMemcpyHostToDevice);

  // Launch n threads in n/512 blocks of 512 threads each
  // Argument passed is a device pointer
  kernel_Sq <<< n/512,512 >>> (d_a);

  // Copy a back into host memory using a helper function
  cudaMemcpy(h_a,d_a,n*sizeof(float),cudaMemcpyDeviceToHost);

  free(h_a); cudaFree(d_a);
}
```


The same program in OpenCL

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>

float *h_a;  // pointer to array in host memory

// The source code of our kernel, stored as an array of strings
const char* OpenCLSource[] = {
        "__kernel void Sq(__global float* a) {",
        "                                      ",
        "      // Index of the element to square  \n ",
        "      // (1D data so dimension index = 0 \n ",
        "      unsigned int i = get_global_id(0).    ",
        "                                            ",
        "      // square it and store back in a   \n ",
        "      a[i] = a[i]*a[i];                     ",
        "}"
};
```

- This source will be compiled on-the-fly and loaded onto the GPU device by our host program, written in C.

```
// Host code - to be compiled with any C compiler, linked against libOpenCL.
int main(int argc, char** argv){

  int n = 4096;
  h_a   = (float *)malloc(n*sizeof(float));

  // Create the same array as in the CUDA example
  int i; for (i=0;i<n;i++) { h_a[i] = (float)(i); }

  // Create a context ignoring any advanced settings / error handling
  cl_context myContext;
  myContext = clCreateContextFromType(0,CL_DEVICE_TYPE_GPU,NULL,NULL,NULL);

  // Query the size of array needed to store the list of devices in
  // the current context, then allocate memory for these and populate.
  size_t ParmDataBytes;
  clGetContextInfo(myContext, CL_CONTEXT_DEVICES,0,NULL,&ParmDataBytes);
  cl_device_id* GPUDevices = (cl_device_id *)malloc(ParmDataBytes);
  clGetContextInfo(myContext,CL_CONTEXT_DEVICES,ParmDataBytes,GPUDevices,NULL);

  // Create a command-queue on the first GPU device
  cl_command_queue GPUCmdQueue;
  GPUCmdQueue = clCreateCommandQueue(myContext,GPUDevices[0],0,NULL);
```



```
// Allocate memory on device for array and populate from the data on the host
cl_mem_flags myFlags = CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR;
cl_mem d_a = clCreateBuffer(myContext,myFlags,n*sizeof(float),h_a,NULL);

// Create OpenCL program from the 9 lines of source code
cl_program myCLProgram;
myCLProgram = clCreateProgramWithSource(myContext,9,OpenCLSource,NULL,NULL);

// Build the program
clBuildProgram(myCLProgram,0,NULL,NULL,NULL,NULL);

// Create a handle to the compiled OpenCL function (Kernel)
cl_kernel OpenCLSq = clCreateKernel(myCLProgram,"sq", NULL);

// Associate the memory we allocated on the device with the kernel arguments
clSetKernelArg(OpenCLSq,0,sizeof(cl_mem),(void*)&d_a);

// At last we can launch the kernel on the GPU
size_t WorkSize[1] = {n}; // 1 dimensional work group of size n
clEnqueueNDRangeKernel(GPUCmdQueue,OpenCLSq,1,NULL,WorkSize,NULL,0,NULL,NULL);

// Copy the result back to host memory
clEnqueueReadBuffer(GPUCmdQueue,d_a,CL_TRUE,0,n*sizeof(float),h_a,0,NULL,NULL);
```



```
// Cleanup
free(GPUDevices);
clReleaseKernel(OpenCLSq);
clReleaseProgram(myCLProgram);
clReleaseCommandQueue(GPUCommandQueue);
clReleaseContext(myContext);
clReleaseMemObject(d_a);
```

This is then compiled as normal with gcc or similar, linking against an OpenCL library. N.B. can link against this from Fortran as well.

```
dq@arnie:> gcc openCL_sq.c -lOpenCL
```

OpenCL is a general purpose method for coding hybrid systems. It is not GPU specific and if therefore necessarily more complicated!

We decided to concentrate on CUDA for simplicity!

CUBLAS

- That last example was essentially a BLAS level 1 operation.
- CUBLAS is an implementation of the BLAS optimised for NVIDIA GPUs.
- Can be invoked from within a CUDA program, or from a standard C code.
- Helper routines are available to initialise the GPU, allocate memory on the GPU, and to copy data between the host and the device.
- Let's revisit the `dgemm` example from Monday's workshop.

CUBLAS – DGEMM

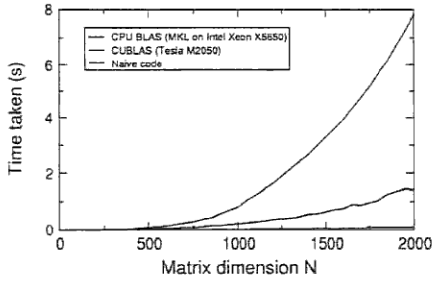
```
#include <cublas.h>  // found in the CUDA SDK

cublasInit();        // initialise CUBLAS

<define matrices a and b in host memory, plus c to hold the result>

// Allocate memory
double *dev_A,*dev_B,*dev_D;
cublasAlloc(N*N,sizeof(double),(void**)&dev_A);
cublasAlloc(N*N,sizeof(double),(void**)&dev_B);
cublasAlloc(N*N,sizeof(double),(void**)&dev_D);

// Put the input matrices onto the GPU device
cublasSetMatrix(N,N,sizeof(double),a,N,dev_A,N);
cublasSetMatrix(N,N,sizeof(double),b,N,dev_B,N);

// Do the matrix multiplication
cublasDgemm('N','N',N,N,N,1.0,dev_A,N,dev_B,N,0.0,dev_D,N);
```


CUBLAS – DGEMM

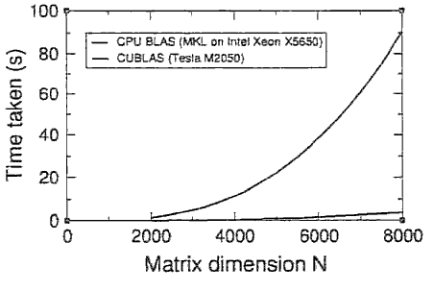
```
// Get the matrix D from the device
cublasGetMatrix(N,N,sizeof(double),dev_A,N,a,N);

// Release memory
cublasFree(dev_A);
cublasFree(dev_B);
cublasFree(dev_D);

// Shutdown CUBLAS
cublasShutdown();
```

- Note the lack of any option to inform CUBLAS if arrays are in row-major or column-major format. Column major only!
- Pointers returned by cublasAlloc are to *device memory* and should never be dereferenced on the host.

CUBLAS – DGEMM



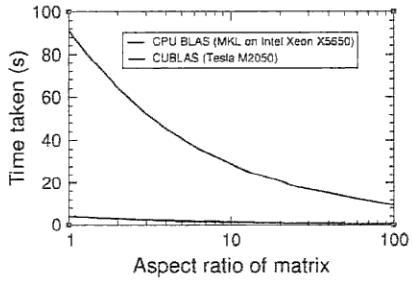
CUBLAS – DGEMM



Don't be a square!

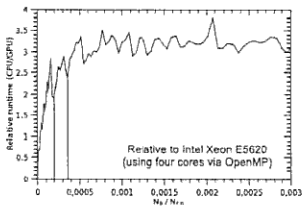
- Recall that in our practical, we never need store or operate on a matrix of size `num_pw` x `num_pw`.
- Largest two matrices we need to multiply together are of dimension `num_pw` x `num_states`.
- In a typical real-world electronic structure calculation we may have 100,000 plane waves (`num_pw`) and 50 electrons (`num_states`) so $O(10^6)$ elements with a 1000:1 aspect ratio.
- The following tests were all performed on matrices with 64 million elements, but varying aspect ratios.

CUBLAS – DGEMM



CUBLAS – CGEMM

- Actual prototype code uses complex data.
- Effectively limited to single precision at the time (Tesla C1060)



- Best case scenario is about a factor of 3.
- This reduces further with double precision.
- ~2.5x for "real" problems.

What's happening?

- For square matrices, blocking algorithms have the greatest benefit due to reuse of data in registers/cache.
- As our matrices get narrower, they approach the limit of one dimensional data (a vector) and the block size becomes small.
- In this limit blocking becomes pointless, and the memory bandwidth feeding our cores is saturated.
- Therefore performance on the GPU suffers. Insufficient bandwidth to feed data to the (many) CUDA cores on the device versus the (single) core on the host.

Miserable speedup

- For realistic problem sizes and dimensions, the orthogonalisation gains very little from running on the GPU.
- Note that it is the memory bandwidth *on the device* which kills speedup, not the bandwidth between the host and the device.
- Can we salvage anything by performing our FFTs on the GPU?
- Remember Amdahl's law..... Hence almost pointless to try.
- Give it a go anyway!

CUFFT

- Like CUBLAS, CUFFT gives us access to the GPU hardware from standard C code.
- Mechanics of using it should be familiar to anyone who has used FFTW.
- Create a plan, and then invoke that plan any number of times to perform the transform.
- Can also perform FFTs in batches, parallelising over transforms within the GPU.

CUFFT – one dimension

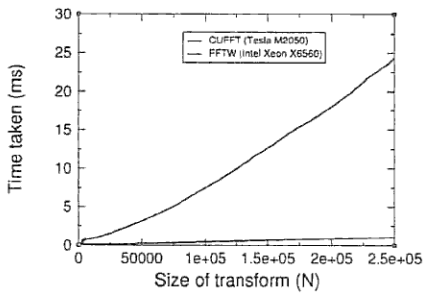
```
include <cufft.h>     // found in the cuda SDK

cufftHandle plan;     // variable to hold the transform plan
cufftComplex *data;   // binary compatible with C99 complex

int N = 1024;         // define the size of the transform

// Allocate memory on the device, don't dereference data on the host.
cudaMalloc((void**)&data, sizeof(cufftComplex)*N);

// Create a 1D FFT plan, for a complex to complex transform
cufftPlan1d(&plan, N, CUFFT_C2C, 1);

// Use the CUFFT plan to transform the signal in place.
cufftExecC2C(plan, data, data, CUFFT_FORWARD);
```


CUFFT – one dimension

```
cufftDestroy(plan); // release memory used by the plan

cudaFree(data);     // free memory on the device
```

- Third argument to `cufftPlan1d` is a constant defined in `cufft.h` to indicate the transform type.
- Note that real to complex and complex to real transforms do not take advantage of the data for extra speedup.
- Final argument to `cufftPlan1d` is the number of transforms to perform in parallel as a batch.

CUFFT – 1D single precision

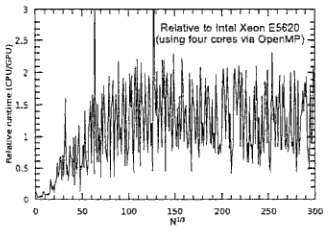


Real crystals live in a 3D world

- Consider a 3D grid of dimensions $n_x \times n_y \times n_z$
- Internals of a 3D FFT reduce to;
  1. $n_y \times n_z$ 1d transforms of length n_x
  2. $n_z \times n_x$ 1d transforms of length n_y
  3. $n_x \times n_y$ 1d transforms of length n_z
- All transforms at steps 1 and 3 can be done in parallel. Step 2 requires the subsets of n_x transforms to be performed in sequence.
- Assume cubic system with `num_pw` = 10^5, n_x=n_y=$n_z \approx 46$

3D FFT, single precision

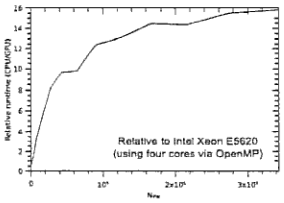
- Data from actual prototype code
- Again limited to single precision at the time (Tesla C1060)



- Unsmoothed.
- Some magic numbers.
- Poor speedup for useful transform sizes.

Final code

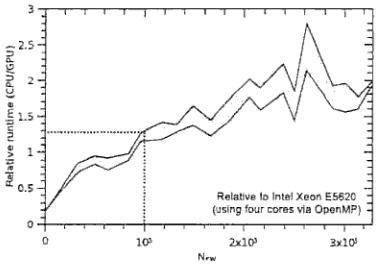
- Beyond using CUBLAS and CUFFT, what else can we do?



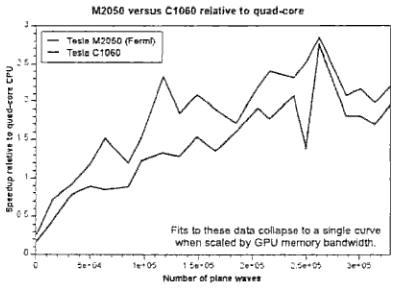
- Custom preconditioning kernel.
- Best speedup of any part of the code.
- Small fraction of overall runtime.

- All wavefunction data (states) are created on the GPU and are never copied to/from the host memory.

Overall code performance



Newer hardware?



Summary

- CUDA, and in particular CUBLAS and CUFFT are a very easy way of using GPU acceleration in many codes!
- Be wary of selective benchmarks!
- A GPU is a parallel computer. Amdahl's law applies. Code can only be as fast as its slowest component!
- NVIDIA are new to the HPC market. They're not fully up-to-speed on the full range of HPC applications.

  [They are however keen to listen, e.g. info on non-square matrices has been fed back via pre-sales.]