Programming GPGPUs

Xinghao Gu
Koushik Ramakrishnan
Venkatarama Subrahmanyan Watrap Raman
Outline

- Introduction to GPUs and GPGPUs
- GPU vs. CPU
- GPU Architecture
- GPU Programming with CUDA and OpenCL
- Bitcoin – mining using GPU
Introduction

• What is GPU?
  ▫ Graphics processing unit
  ▫ Built for accelerating the creation of images
  ▫ Modern GPUs are very efficient at manipulating computer graphics
  ▫ highly parallel structure
What is GPGPU?

- General-purpose computing on graphics processing units
- Using the GPU for General Purpose Calculations
GPU History (1980s)

- Intel iSBX 275 Video Graphics Controller Multimodule Board
  - Accelerating drawing lines, arcs, rectangles, and character bitmaps
- Texas Instruments released first microprocessor with on-chip graphic capabilities
GPU History (1990s)

• By 1995, all major PC graphics chip makers had added 2D acceleration support to their chips.

• In 1999, Nvidia popularized the term ‘GPU’ and released ‘the world’s first GPU’ GeForce 256.
  ▫ processing a minimum of 10 million polygons per second
GPU History (1990s)

- Professional API
  - OpenGL
    - appeared in the early '90s as a professional graphics API
  - DirectX
    - became popular among Windows game developers during the late 90s
Early GPUs: fixed function pipeline
Early GPUs: fixed function pipeline

- Receives triangle data
- Converts them into a form that hardware understands
- Store the prepared data in vertex cache
Early GPUs: fixed function pipeline

- Vertex shading, transform, and lighting
- Assigns per-vertex value

Diagram showing the fixed-function pipeline with steps like Host interface, Vertex control, VS/T & L, Triangle setup, Raster, Shader, ROP, FBI, Vertex cache, Texture cache, Frame buffer memory.
Early GPUs: fixed function pipeline

Fixed-Function Pipeline

Creates edge equations to interpolate colors across pixels touched by the triangle
Early GPUs: fixed function pipeline
Early GPUs: fixed function pipeline
Early GPUs: fixed function pipeline

The raster operation: performs color raster operations that blend the color of overlapping objects for transparency and antialiasing.

Examples of anti-aliasing operations: (a) triangle geometry, (b) aliased, and (c) anti-aliased.
Early GPUs: fixed function pipeline

Fixed-Function Pipeline

The frame buffer interface manages memory reads/writes.
GPU History (early 2000s)

- the instruction set of VS/T&L stage was exposed to developers
- General programmability extended to shader stage
GPU History (later 2000s)

- Till 2006, graphics chips were very difficult to use for computing because programmers had to use graphics API to express non-graphics computations.
- In 2007, CUDA facilitate the ease of parallel programming
- OpenCL developed by Apple has become widely supported
## GPU development

<table>
<thead>
<tr>
<th>Date</th>
<th>Product</th>
<th>Transistors</th>
<th>CUDA cores</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>RIVA 128</td>
<td>3 million</td>
<td>—</td>
<td>3D graphics accelerator</td>
</tr>
<tr>
<td>1999</td>
<td>GeForce 256</td>
<td>25 million</td>
<td>—</td>
<td>First GPU, programmed with DX7 and OpenGL</td>
</tr>
<tr>
<td>2001</td>
<td>GeForce 3</td>
<td>80 million</td>
<td>—</td>
<td>First programmable shader GPU, programmed with DX8 and OpenGL</td>
</tr>
<tr>
<td>2002</td>
<td>GeForce FX</td>
<td>125 million</td>
<td>—</td>
<td>32-bit floating-point (FP) programmable GPU with Cg programs, DX9, and OpenGL</td>
</tr>
<tr>
<td>2004</td>
<td>GeForce 6800</td>
<td>222 million</td>
<td>—</td>
<td>32-bit FP programmable scalable GPU, GPGPU with Cg programs, DX9, and OpenGL</td>
</tr>
<tr>
<td>2006</td>
<td>GeForce 8800</td>
<td>681 million</td>
<td>128</td>
<td>First unified graphics and computing GPU, programmed in C with CUDA</td>
</tr>
<tr>
<td>2007</td>
<td>Tesla T8</td>
<td>681 million</td>
<td>128</td>
<td>First GPU computing system programmed in C with CUDA</td>
</tr>
<tr>
<td>2008</td>
<td>GeForce GTX 280</td>
<td>1.4 billion</td>
<td>240</td>
<td>Unified graphics and computing GPU, IEEE FP, CUDA C, OpenCL, and DirectCompute</td>
</tr>
<tr>
<td>2008</td>
<td>Tesla T10, S1070</td>
<td>1.4 billion</td>
<td>240</td>
<td>GPU computing clusters, 64-bit IEEE FP, 4-byte memory, CUDA C, and OpenCL</td>
</tr>
<tr>
<td>2009</td>
<td>Fermi</td>
<td>3.0 billion</td>
<td>512</td>
<td>GPU computing architecture, IEEE 754-2008 FP, 64-bit unified addressing, caching, ECC memory, CUDA C, C++, OpenCL, and DirectCompute</td>
</tr>
</tbody>
</table>
Why GPGPU?

- Since 2003, due to energy consumption and heat dissipation issues that limited the increase of the CPU clock rate
- GPUs have much more cores than CPUs
  - CPUs now have 2, 4, 8...processors
  - GPUs now have 32, 64, 128...processors
- Over 30 years evolution, GPUs have been changed from a simple pipeline to highly parallel design.
Birth of GPGPU

Step 1: Designing high-efficiency floating-point and integer processors.

• Step 2: Exploiting data parallelism by having large number of processors

• Step 3: Shader processors fully programmable with large instruction cache, instruction memory, and instruction control logic.

• Step 4: Reducing the cost of hardware by having multiple shader processors to share their cache and control logic.

• Step 5: Adding memory load/store instructions with random byte addressing capability

• Step 6: Developing CUDA C/C++ compiler, libraries, and runtime software models.
CPU vs GPU
CPU vs GPU

• 2 Design Philosophies

• Maintain execution speed of old sequential programs → CPU

• Increase throughput of parallel programs → GPU
CPU vs GPU (Architecture)

FIGURE 1.1
CPUs and GPUs have fundamentally different design philosophies.
CPU vs GPU

• CPU is latency-oriented designed
  ▫ Large cache memory
  ▫ Low-latency arithmetic units
  ▫ Sophisticated control logic

• GPU is parallel, throughput-oriented computing engines
Limitations of GPGPUs

- The parallelizable portion of the code
- The communication overhead between CPU and GPU
Modern GPU Architecture

**GeForce 8800 (2007)**

16 highly threaded SM's, >128 FPU's, 
367 GFLOPS, 768 MB DRAM, 
86.4 GB/S Mem BW, 
4GB/S BW to CPU

Diagram showing the architecture of the GeForce 8800 GPU with various components and connections.
Modern GPU Architecture

SPs within SM share control logic and instruction cache

Graphic Double Data Rate DRAM (GDDR)
Modern GPU Architecture

- Much higher bandwidth than typical system memory
- A bit slower than typical system memory
- Communication between GPU memory and system memory is slow
Programming Concepts

- OpenCL - portable standard adopted by many vendors (AMD, Intel, IBM, NVIDIA)
  - platform independent

- CUDA - parallel computing platform from NVIDIA
  - can result in faster execution than OpenCL
CUDA Programming Model: A Highly Multithreaded Coprocessor

• The GPU is viewed as a compute device that:
  ▫ Is a coprocessor to the CPU or host
  ▫ Has its own DRAM (device memory)
  ▫ Runs many threads in parallel

• Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads

• Differences between GPU and CPU threads
  ▫ GPU threads are extremely lightweight
    • Very little creation overhead
  ▫ GPU needs 1000s of threads for full efficiency
    • Multi-core CPU needs only a few
Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...

Courtesy: NDVIA
Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads

Courtesy: NDVIA
Transparent Scalability

- Hardware is free to assigns blocks to *any* processor at *any* time
  - A kernel scales across *any* number of parallel processors

Each block can execute in any order relative to other blocks.
CUDA Device Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object

- **cudaFree()**
  - Frees object from device **Global Memory**
    - Pointer to freed object
CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device

![CUDA Memory Diagram](image_url)
void vecAdd(float *A, float *B, float *C, int N) {
    for(int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}

int main() {
    int N = 4096;
    float *A = (float *)malloc(sizeof(float)*N);
    float *B = (float *)malloc(sizeof(float)*N);
    float *C = (float *)malloc(sizeof(float)*N);

    init(A); init(B);

    vecAdd(A, B, C, N);
    free(A); free(B); free(C);
}

// Computational Kernel

// Allocate Memory

// Initialize Memory

// De-allocate memory
Vector Addition (GPU)

```c
__global__ void gpuVecAdd(float *A, float *B, float *C)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
}
```

GPU Computational kernel
Vector Addition (GPU)

```c
int main() {
    int N = 4096;
    float *A = (float *)malloc(sizeof(float)*N);
    float *B = (float *)malloc(sizeof(float)*N);
    float *C = (float *)malloc(sizeof(float)*N);
    init(A); init(B);
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, sizeof(float)*N);
    cudaMalloc(&d_B, sizeof(float)*N);
    cudaMalloc(&d_C, sizeof(float)*N);
    cudaMemcpy(d_A, A, sizeof(float)*N, HtoD);
    cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
    dim3 dimBlock(32,1);
    dim3 dimGrid(N/32,1);
    gpuVecAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);
    cudaMemcpy(C, d_C, sizeof(float)*N,DtoH);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(A);
    free(B);
    free(C);
}
```

Allocate memory on GPU

Initialize memory on GPU

Configure threads

Run kernel (on GPU)

Copy results back to CPU

Deallocate memory on GPU
Programming Model: Square Matrix Multiplication Example

- $P = M \times N$ of size $\text{WIDTH} \times \text{WIDTH}$
- Without tiling:
  - One thread handles one element of $P$
  - $M$ and $N$ are loaded $\text{WIDTH}$ times from global memory
Step 1: Matrix Data Transfers

// Allocate the device memory where we will copy M to Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size,
cudaMemcpyHostToDevice);

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size,
cudaMemcpyDeviceToHost);
...

// Free device memory
cudaFree(Md.elements);
Step 2: Matrix Multiplication
A Simple Host Code in C

// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i)
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];
                double b = N.elements[k * N.width + j];
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
}
int main(void) {
    // Allocate and initialize the matrices
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
    return 0;
}
Step 3: Matrix Multiplication
Host-side code

// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);
    CopyToDeviceMatrix(Pd, P); // Clear memory
Step 3: Matrix Multiplication
Host-side Code (cont.)

// Setup the execution configuration
dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

// Read P from the device
CopyFromDeviceMatrix(P, Pd);

// Free device matrices
FreeDeviceMatrix(Md);
FreeDeviceMatrix(Nd);
FreeDeviceMatrix(Pd);
Step 4: Matrix Multiplication
Device-side Kernel Function

// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
Step 4: Matrix Multiplication
Device-Side Kernel Function (cont.)

for (int k = 0; k < M.width; ++k)
{
    float Melement = M.elements[ty * M.pitch + k];
    float Nelement = Nd.elements[k * N.pitch + tx];
    Pvalue += Melement * Nelement;
}

// Write the matrix to device memory;
// each thread writes one element
P.elements[ty * P.pitch + tx] = Pvalue;
Outline

• Introduction to OpenCL – Overview of OpenCL, its architecture and model
• Introduction to Bitcoin – Overview of what a Bitcoin is how it works
• An example application of using OpenCL – Bitcoin Mining
Introduction to OpenCL

• What is OpenCL?
  ▫ A framework for writing programs that execute across heterogeneous platforms consisting of CPU, GPU, DSP and others.
  ▫ Built for portability among devices.
  ▫ Maintained by non-profit Khronos Group.
  ▫ Adopted by Apple, Intel, Qualcomm, AMD, Nvidia, ARM Holdings and other major hardware players.
What is OpenCL

• OpenCL provides a specification that includes a language (based on C99) for programming.
• Constantly under development, latest version is OpenCL 2.0 – previous versions include OpenCL 1.0, 1.1, and 1.2.
• Heterogeneity means support for both data-parallel (GPU) and task-parallel (CPU) computing.
Anatomy of OpenCL

- Platform layer API
- Runtime API
- Language Specification
A HW abstraction layer over diverse computational resources.

Allows the programmer to,

- Query, select and initialize *compute devices*.
- Create *compute context* and *work-queues.*
Runtime API and Language specification

• Runtime API
  ▫ Allows the developer to queue up *compute kernels* for execution and is responsible for managing scheduling, the compute and memory resources.

• Language Specification
  ▫ Based on C99, omits many features of C99 like function pointers, bit fields, variable length arrays etc.
  ▫ These are omitted because they cannot help conform to the OpenCL universe.
Models in the OpenCL abstraction

- These models define the OpenCL architecture.
- They provide the abstraction that is standard on all the devices and enables the developer develop for this abstraction.
- This enables portability.
- Models in the OpenCL system,
  - OpenCL Platform Model.
  - OpenCL Execution Model.
  - OpenCL Memory Model.
OpenCL Platform Model

- The platform is made up of a Host which controls one or more Compute Devices.
- These Compute Devices each have several compute units.
- Ex: ATI Radeon HD 5870,
  - 20 Compute Units with 80 Processing Elements/compute unit
This equals 1600 physical processing units.
OpenCL Execution Model

- Applications run on a host from which Work is submitted to Compute Devices.
- *Compute Kernel* is the basic unit of work or executable code. Analogous to a C function.
- These are submitted to the compute devices through queues.
- A work-item is the basic unit of work Say, like addition of two numbers.
- The kernel is the code for a work-item. Context is the environment a work-item executes - the devices, their memory, the queues.
OpenCL Execution Model

- Execution takes place on an N-Dimensioned Index space.
- Kernels can be executed across a domain of work-items.
- Work-items grouped into work-groups.
- Synchronize between work-groups and not outside.
- Here, Global Space – 1024*1024
  - Local Space – 128*128
OpenCL Memory Model

• A multi-level model with memory ranging from private memory visible only within work-items (compute-units) to global memory visible to all work-groups.
• Memory management is explicit, meaning programmer responsible for moving data, from host -> global -> local ... and back
OpenCL Compilation Model

• OpenCL uses Dynamic/Runtime compilation model
• The code is compiled to an Intermediate Representation (IR)
  ▫ Usually to a virtual machine
  ▫ Known as offline compilation
• The IR is compiled to a machine code for execution.
  ▫ This step is much shorter.
  ▫ It is known as online compilation.
Introduction to Bitcoin

• Is a cryptocurrency.
• What is cryptocurrency?
  ▫ A form of currency taking the digital form which is,
    • Decentralized
    • Peer-To-Peer
    • Digital
  ▫ Use of cryptography to generate currency, validate and secure transactions.
  ▫ May (usually don’t) provide anonymity
Motivations for Bitcoin

• Payments to be direct.
• Reduce trust among each other.
• Reduce a third-party centralized system for regulation of transfer of funds and transactions.
• Prevent currency crimes
• Problem of direct payments already solved by use of Digital Signatures.
• How to prevent double-spending?
Digital Signature - is an electronic signature that can be used to authenticate the identity of the sender of a message or the signer of a document.

A digital signature of a message is obtained by encrypting the hash of the message with the private key.

Now anyone can verify that the message came from you and is intact by decrypting the message using the private key and verifying the hash of the message.

A cryptographic hash of a message is a one-way function that generates a fixed-set of bits that represents the message.
Elements of Bitcoin

- Bitcoin network is a collection of nodes, where each node can be a mining node, a sender and a receiver.
- Transactions are how Bitcoins are transferred from one client to another.
- Mining is the process of confirming a transaction and adding to a Block chain. Mining also serves the purpose of Bitcoin creation.
- A Block chain is the public, global record chain of all the transactions made so far. It can be imagined as a universal, publicly available chain of ledger books where each ledger book is mathematically related to its previous one chronologically.
The Bitcoin and the wallet

- A Bitcoin is a public and private-key pair. The public key represents the coin itself. A Bitcoin is referred to as 1BTC.
- A Bitcoin wallet is a file with its own private-key which stores a set of private-keys which represents the Bitcoins you have received and the denomination of them.
- Only the private-keys are stored.
- A Bitcoin wallet should be stored in a secure way.
- It represents your/client’s address.
- Don’t lose the wallet.
Transactions

- A Bitcoin is put into circulation by spending it.
- Spending a Bitcoin means performing a transaction.
- A transaction maintains the history of spending of a Bitcoin.
- A transaction is made when the current owner digitally signs the hash of the previous transaction and the public key of the next owner and adding these to the end of the coin.
How to prevent double-spending?

• What is double-spending?
  ▫ Because this is *digital* currency, there is no tangible physicality that prevents it from being spent twice.

• We only consider the earliest transaction to be the one that counts.

• To accomplish this in a decentralized fashion, we have to make a public announcement and need a system of participants to agree that this was the first received.

• The payee needs proof that at the time the transaction was made, the majority of the people agreed that it was the first received.
Timestamp Server

- A timestamp server works by taking a hash of a block of items to be time stamped and widely publishing the hash where people can see and agree.
- The timestamp proves that the data must have existed at that time, obviously in order for the hash to be created.
- Additionally, the hashes created include the timestamp of the previous block hence reinforcing the previous transactions, forming a chain.
- Instead of publishing this transaction onto physical medium, we follow a proof-of-work system to implement a distributed time stamp server.
Proof-of-work system

- A proof-work-system is a system designed to prevent denial of service attacks.
- The proof-work-system serves to prove that the request made is legitimate and is worth considering or serving.
- Example of proof-of-work systems include, CAPTCHAs, exams etc.
- A proof-work-system used in Bitcoin is the Hashcash system where we are required to calculate a partial hash value.
- This calculation involves calculating a hash value that is less than or equal to a target hash value.
Proof-of-work system

- This target value is calculated periodically based on the “hash power” of the mining network to keep the number of Bitcoins generated a constant value.
- The target hash is calculated to keep the average time to find a hash at 10mins by the whole network.
- This process is called Bitcoin mining, because a Bitcoin is “mined” by hard computational work which involves finding a target hash value.
- This work is hard because, the problem time is exponential to the number of zeroes in the beginning of the target hash value. So, if the time needs to be stretched the target hash value is lower.
Proof-of-work-system

- When the hash for a block is found, it is broadcasted throughout the network and is validated by the nodes when they find that the hash matches the data provided.
- This block is then added to the chain and the processing for the next block begins.
- This required hash value is found on a trial and error basis by using a nonce and calculating the hash value, finding if its valid and incrementing it to calculate further.
- This is where GPGPU comes into picture.
- Data-parallelism exploited.
Bitcoin Mining Hardware

• Bitcoin mining hardware timelines from using general CPU and has come a long way in being very efficient and powerful, using special hardware such as a GPU to using ASIC – Application Specific Integrated Circuit, which are hardware created only to mine Bitcoins.
• The power of a hardware is measured in Hashes per second and Giga hashes per second.
• Matured from CPU -> GPU -> FPGAs -> ASIC
• Most efficient GPU for Bitcoin mining was the ATI Radeon HD 5970.
Bitcoin mining rigs
Bitcoin Mining using OpenCL

- Exploiting the data-parallelism that is present, there are codes for Bitcoin mining using OpenCL.
- It was very popular because anyone can run OpenCL code whether or not they had GPU or nor. Can be run on multi-core computers as well.
- Cheap alternative to Bitcoin hunting instead of massive investment.
- ASIC are very power efficient and still process huge amounts of hashes per second.
- This has reduced Bitcoin mining as an attractive market for making money. So collect transaction fees for mining. (The default when all the 21Billion Bitcoins are mined).
Detect the platform,
   //get the platformids and the number of platforms
   clGetPlatformIDs(0,NULL,&numplatforms);
   printf("%d OpenCL platforms found\n",numplatforms);

Detect the devices that are in the specified platform m_platform
   cl_platform_id *pids;
   pids=new cl_platform_id[numplatforms];
   clGetPlatformIDs(numplatforms,pids,NULL);
   clGetDeviceIDs(pids[m_platform],CL_DEVICE_TYPE_GPU,0,NULL,&m_devicecount);
   printf("%d OpenCL GPU devices found on platform %d\n",m_devicecount,m_platform);
Set the device to be used

- `m_device=devices[m_deviceindex];`
- `printf("Setting OpenCL device to device %d\n",m_deviceindex);`
- `clGetDeviceIDs(pids[m_platform],CL_DEVICE_TYPE_GPU,1,&m_device,NULL);`

Create the context and command queue,

- `m_context=clCreateContext(0,1,&m_device,NULL,NULL,&rval);`
- `printf("Create context rval=%d\n",rval);`
- `m_commandqueue=clCreateCommandQueue(m_context,m_device,0,&rval);`
- `printf("Create command queue rval=%d\n",rval);`
Create the program object for the specified context and build the program for the GPU device,

- `std::string srcfile=ReadFileContents("bitcoinmineropencl.cl");`
- `char *src=new char[srcfile.size()+1];`
- `printf("Creating program with source\n");`
- `m_program=clCreateProgramWithSource(m_context,1,(const char **)&src,0,&rval);`
- `printf("Building program with options %s\n",buildoptions.c_str());`
- `rval=clBuildProgram(m_program,1,&m_device,buildoptions.c_str(),0,0);`
- `printf("Build program rval=%d\n",rval);`
Create the kernel,
\[\text{m\_kernel = clCreateKernel(m\_program,"opencl\_process",rval);}\]
\[\text{printf("Create kernel rval=%d\n",rval);}\]
Allocate the resources,
\[\text{m\_in = (opencl\_in *)malloc(sizeof(opencl\_in));}\]
\[\text{m\_out = (opencl\_out *)malloc(numb*numt*sizeof(opencl\_out));}\]
\[\text{m\_devin = clCreateBuffer(m\_context,CL\_MEM\_READ\_ONLY,sizeof(opencl\_in),0,0);}\]
\[\text{m\_devout = clCreateBuffer(m\_context,CL\_MEM\_READ\_WRITE,numb*numt*sizeof(opencl\_out),0,0);}\]
Set the kernel arguments,
- `const cl_uint loops = GetStepIterations();`
- `const cl_uint bitshift = GetStepBitShift() - 1;`
- `clSetKernelArg(m_kernel, 0, sizeof(cl_mem), (void *)&m_devin);`
- `clSetKernelArg(m_kernel, 1, sizeof(cl_mem), (void *)&m_devout);`
- `clSetKernelArg(m_kernel, 2, sizeof(cl_uint), (void *)&loops);`
- `clSetKernelArg(m_kernel, 3, sizeof(cl_uint), (void *)&bitshift); // bitshift equals the number of zeroes to be checked`

Queue the kernel for execution,
- `const unsigned int cnumt = m_numt;`
- `const unsigned int dim = m_numt * m_numb;`
- `clEnqueueNDRangeKernel(m_commandqueue, m_kernel, 1, 0, &dim, &cnumt, 0, 0, 0);`

Retrieve the results and transfer it back to host,
- `clEnqueueReadBuffer(m_commandqueue, m_devout, CL_TRUE, 0, m_numb * m_numt * sizeof(opencl_out), m_out, 0, 0, 0);`

Kernel code has the code for finding the hash with a nonce and if the found hash value equals zero, return the best nonce.
Reference

- Wikipedia
- [http://www.geforce.com/hardware](http://www.geforce.com/hardware)
- Slides by [Intro to GPGPU with CUDA](http://www.geforce.com/hardware) by Rob Gillen
- [http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clEnqueueNDRangeKernel.html](http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clEnqueueNDRangeKernel.html)
- [http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clEnqueueReadBuffer.html](http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clEnqueueReadBuffer.html)