-->

CUDA PASSWORD CRACKING

In this program we are exploring ways of cracking passwords using NVIDIA's
CUDA(Compute Unified Device Architecture) API.

CUDA is a general purpose parallel computing platform and Programming model
that uses the GPU's parallel processing units or as NVIDIA calls them CUDA
Cores in order to execute many more complex computational problems than
multithreading on the CPU could handle.

We decided on this method because it enables to utilize much more computational
power on the many core GPU than the multicore CPU would be able to handle and allow us to work on
this project in a bigger scale than if we had just used the CPU.

Programming Model

Within the context of CUDA terms such as `host` and `device` more or less mean the CPU and GPU

CUDA C++ provides users of the API to more easily write programs for execution
by the device (GPU) it consists of a minimal set of extensions to the C++
language and a runtime library any file containing any of these extensions or
library calls also must be compiled with NVIDIA's `nvcc` compiler

Some of the basic extensions include

- Function Specifiers
    - `__global__`: A kernel that is executed on the `device` and can be called from either `device` or `host`
    - `__host__`: A function that is executed on the `host` and is only callable from the `host`
    - `__device__`: A function that is executed on the `device` and is only callable from the `device`

- Memory Specifiers
    - `__device__`: mutable global memory space
    - `__constant__`: constant global memory space
    - `__shared__`: mutable shared block memory space

Kernel

CUDA C++ extends C++ by allowing the programmer to define C++ functions called `kernels`, that when executed are run many times over in parallel as opposed to only once like a normal C++ function.

The CUDA `kernel` represents your starting point when you are parallelizing your tasks onto the GPU and it requires the `__global__` declaration specifiers so it can be called from the `host` and executed on the `device`

In order to call a kernel from the host you need to use the special CUDA extension syntax `<<<...>>>` each thread that executes the kernel is given a unique thread ID that is accessible through the built-in variables such as `threadIdx`

The code below will execute N threads that print their id to the terminal

```
// Kernel Definition
__global__
void Example() {
    printf("Hello from thread %i",threadIdx.x); // threadIdx is a builtin variab
}

int main() {
    // Kernel is executed with N threads parallelized
    Example<<<1,N>>>();
    cudaDeviceSynchronize();
}
```

Thread Hierarchy

For convenience `threadIdx` is a 3-component vector so that threads can be identified in one/two/three dimension thread index forming a one/two/three dimension block of threads, called a *thread block*. Thread blocks provide a more natural way of invoking computation across elements such as a vector or matrix

To declare a thread block with these dimensions you need to make a `dim3` variable and specify the number of threads per axis you need

```
__global__
void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Memory

CUDA threads may access data from multiple spaces during their execution. Each thread has private local memory, each thread block has shared memory visable to all threads of the block, and there is also global and global constant memory space available to any thread.

Memory Hierarchy

- per thread local memory
- per block shared memory
- per block cluster shared memory
- global memory shared between all GPU kernels

Since the `host` and `device` have seperate memory spaces CUDA provides API functions to allocate, deallocate, and copy device memory. Memory is typically allocated with `cudaMalloc()` and freed with using `cudaFree()` and you can do memory transfers from between the `host` and `device` with `cudaMemcpy()`

The code below provides an example of allocating memory on the `device` in order to add 2 vectors together

```
__global__
void VecAdd(float* A, float* B, float* C, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Passwords Cracking

For simplicity our password cracking algorithms are focused on generating as many different passwords and doing a simple string compare against the known password

Dictionary (from_list.cu)

We load a dictionary of English words and copy it to the GPU. Then, for every pair of words, we spawn a thread to try 8 variations of the password (different separators and capitalization). This means we test all possible two-word combinations with 8 variants, which totals 24,294,251,592 unique threads all tested in parallel using CUDA finishing in about 2 seconds.

Most Common Passwords

We load a list of the 100,000 most commonly used passwords and copy it to the GPU. Then, we spawn one thread per password, and each thread compares its assigned password with the target. This means all 100,000 passwords are checked in parallel using CUDA, allowing us to test the entire list almost instantly.

Brute Force

Since the brute force algorithm too a lot longer compare to the others we decided to limit the algorithm to only try 8 length passwords of lowercase letters and numbers totalling 36 unique characters

We brute force every possible combination of these 36 characters to a string by essentially converting the unsigned long long id of each thread to a string by treating the id as a number is base 36 representation which would give us a password to check against for every single id we generate using CUDA.

A similar example of how i make a string from the id would be to only focus on a id with numbers and since there are only 10 number characters its base 10 and i would only need to assign character '0' to value 0 and character '9' to value 9 and create a string based from that information so that id (012345678) would be string "012345678"

References

CUDA Documentation