barREUcuda

Abdulmajed Dakkak

June 18th, 2008

The CUDA programming language offers new opportunities to a field regarded by many to be fairly difficult — parallel programming. Through a simple framework that comparable to Google's recent (over hyped) **map reduce** frame work, we can identify certain components in a serial program and parallelize repetitive function calls.

The concept behind CUDA is fairly simple. You break your algorithm into small pieces and hand each thread a little piece of computation. And, since most algorithms have a while loop embedded in them, the code inside the while loop can be the code handed to each thread. If that loop is repetitive, i.e. there is little difference between them, then we can gain a performance boost, since the subroutines are called all in one clock cycle (or `warp`).

CUDA accomplishes this by extending (only a bit) `C`. They add declarations that define whether a function runs on the device (the `GPU`) or the host (the `CPU`). They also add new types of memory which need to be studied more carefully.

Take, for example, quicksort, which is defined by the following `python/functional` code.

```
def quicksort(m):
    pivot = m[0]
    p = filter(lambda x: x <= pivot, m)
    p = filter(lambda x: x >  pivot, m)
    return quicksort(p) + [pivot] + quicksort(q)
```

The above implementation does not sort in place, thus requiring $O(n)$ size. The following is a `C` implementation that sorts in place.

```
void qsort(float* m, int start, int end)
{
    int i;
    int j = start;
    float pivot = m[start];
    for(i = start+1; i < end; i++)
    {
        if(m[i] < pivot)
            m[j++] = m[i]
    }
    m[j] = pivot;
    qsort(m, start, j-1);
    qsort(m, j+1, end);
}
```

Recursion in an algorithm is a big hint that the algorithm can be parallelized — even though CUDA's function cannot recursed. Most algorithms can be easily parallelized. This is specifically the case for programs that simulate a large quantity of relatively independent objects. It so happens that most physics can be reduced to simulating many particles' interactions with the environment, and many abstractions exist to simulate this interaction. Take 1D rule based cellular automata, for example. In a serial program on would keep track of the previous row and would write the following `C` function to compute the next row

```
while(row_count < MAXROWS)
{
    for(i = 0; i < ROWLENGTH; i++)
    {
        if(i == 0)
        {
            row[i] = rule(0, old_row[i], old_row[i+1])
        }
        else if(i == ROWLENGTH-1)
        {
            row[i] = rule(old_row[i-1], old_row[i], 0)
        }
        else
        {
            row[i] = rule(old_row[i-1], old_row[i], old_row[i+1])
        }
    }
    tmp = old_row
    old_row = row
    row = tmp
}
```

A CUDA version of the previous program might look like the following

```
__device__ void rule(int left, int top, int right)
{
    // insert automata rule here
    ...
}

__global__ void kernel(int* old_row, int* row)
{
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

    if(i == 0)
    {
        row[i] = rule(0, old_row[i], old_row[i+1])
    }
    else if(i == ROWLENGTH-1)
    {
        row[i] = rule(old_row[i-1], old_row[i], 0)
    }
    else
    {
        row[i] = rule(old_row[i-1], old_row[i], old_row[i+1])
    }
}

void run( )
{
    dim3 block(16, 16);
    dim3 grid(ROWLENGTH / block.x, ROWLENGTH / block.y);
    while(row_count < MAXROWS)
    {
        kernel<<< grid, block>>>(old_row, row);
        ...
        display(row);
    }
    ...
}
```

The above code will easily achieve more than 20x speedup, but with some optimization, it can run much much faster. A similar program can be sketched for 2D cellular automata such as the Game of Life, and a 3D cellular automata could also be implemented (when I find an example of 3D cellular automata).

The CUDA SDK provides a variety of examples exposing the power of parallel processing. CUDA's N-Body problem, for example, simulates tens of thousands of particles using the direct method which calculates the force between two particles by a slight variation of Newton's 2-Body formula

$\vec{F}_{ij} = G\frac{m_i m_j}{|\vec{r}|^2+\epsilon^2} \bullet \frac{\vec{r}}{|\vec{r}|}$

Such algorithm is unfeasible on commodity CPU's, and most physicist have used a computationally feasible, albeit undesired, algorithm for the n-body problem. CUDA changed the landscape, making supercomputing more "democratic."

Other examples in the SDK include `fluidsGL` which simulates fluid dynamics based on the idea of stable fluids. It too achieves speed that is unattained on a CPU. On the downside, however, the program is in 2D, and relatively uninteresting for 3D visualization.

In this project we would like to bring the power of the CUDA chip to the CAVE/CUBE. There are a few reasons to do that:

1. **Price** — CUDA enabled chips are reasonably priced and are within the reach of a consumer. The lower end chips are under `100` dollars, while the higher end ones go for over `1,000` dollars.
2. **Abundance** — NVIDIA claims to have sold 40 million cards that are CUDA enabled.
3. **Power** — While price and abundance are important to guarantee that this is not just a hype, the performance of the CUDA chip makes the case for its importance. A CUDA enabled GPU can easily outperform a CPU for half the price.

The CUDA chip is not reserved just for visualization. Two recent papers[1] [2] implemented the AES cryptography algorithm with considerable speedup. Other people used the power provided by CUDA for more sinister purpose[3] . By porting the `md5crack` program to CUDA, the author was able to crack all md5 passwords less than 8 characters by brute force in under 16 minutes.

Take, for example, the University of Antwerp in Belgium which recently built a super computer with four NVIDIA GPUs for under `4000` dollars. The system, which is contained in one desktop workstation, can outperform a cluster of hundreds of machines.

[1]Yamanouchi, T.: AES Encryption and Decryption on the GPU. In: GPU Gems 3, Addison-Wesley Professional, Reading (2007)

[2]S.A. Manavski. CUDA COMPATIBLE GPU AS AN EFFICIENT HARDWARE ACCELERATOR FOR AES CRYPTOGRAPHY.

[3]Notes: Cuda md5 hashing experiments. http://majuric.org/software/cudamd