TR-2014-08

The Interplay Between NVIDIA's `Thrust` GPU Computing Template Library and Unified Memory Support in CUDA 6.0

Ang Li, Radu Serban, Dan Negrut

June 4, 2014

Abstract

The most important feature that CUDA 6.0 brings to bear is the implementation of a Unified Memory model [5]. This work investigates the interplay between this CUDA 6.0 feature and several functions from the `Thrust` library [3].

Contents

1 Introduction

`Thrust` [3] is a powerful and versatile library of parallel algorithms and data structures. It provides a flexible, high-level interface for GPU programming that greatly enhances developer productivity. Using `Thrust`, C++ developers can write a few lines of code to perform GPU-accelerated `sort`, `scan`, `transform`, and `reduction` operations at times one order of magnitude faster than multi-core CPU solutions.

The release of CUDA 6 [2] brings along one of the most significant changes in the almost decade-long history of CUDA: the support for Unified Memory [5]. Unified Memory creates a pool of managed memory that is shared between the CPU and GPU thus bridging the CPU-GPU divide. Data stored in managed memory is accessible to both the CPU and GPU through single pointer use. Under the hood, the system automatically migrates data allocated in Unified Memory between host and device so that it appears CPU memory to code running on the CPU and device memory to code running on the GPU. For now, this migration takes place across the PCI-Express bus. In other words, the CPU to/from GPU data movement is as fast as it used to be. The only advantage is that this happens transparently to the user. Gone are in this case explicit data movement calls, which leads to tighter and neater code that is easier to produce and less prone to coding bugs.

Herein, we report early results related to an analysis carried out to gauge the interplay between `Thrust` functions and the new Unified Memory model. The motivation for this study is the fact that `Thrust` has no explicit support for the "Unified Memory" space; i.e., one cannot create something like managed vectors. Instead, the user can

- Directly pass unified memory raw pointers to `Thrust`'s function calls and explicitly specify the parallel execution policy [1]. This approach requires very few lines of codes. Unfortunately, the `Thrust` library does not seem to fully support raw pointers as far as execution on the device is concerned. This seems to be an issue independent of the concept of Unified Memory.

- Right before calling a `Thrust` function, a unified memory raw pointer is wrapped as a `Thrust` device pointer. From that point on, `Thrust` handles the vector as any other vector regardless whether the wrapped array is in managed or un-managed memory.

In our test, we studied several `Thrust` function calls. The objective was to understand whether we can pass a raw or wrapped-up managed memory pointer to a `Thrust` function call. Note that we use both the terms "unified memory raw pointer" and "managed memory raw pointer" These therms both refer to a pointer that points to managed memory and which can be dereferenced both from the CPU and GPU. Given the current trend of integrating the CPU and GPU on the same die and the fact that they'll be sharing the same unified memory, we prefer the term "unified memory raw pointer" to "managed memory raw pointer" since the concept of "managed memory" will most likely disappear as the "managing" will be totally transparent to the user.

2 Test Results

Table 1 summarizes our findings. An × indicates that the function call failed to produce the correct outcome. A successful call is marked by √. The code used to generate these results is available on GitHub [4].

Op. Type	**Func. Name**	Host [H]				Device [D]			
		H-R	H-W	H-M-R	H-M-W	D-R	D-W	D-M-R	D-M-W
Fill	`fill`	√	√	√	√	√	√	√	√
Copy	`copy`	√	√	√	√	√	√	√	√
Transformation	`transform`	√	√	√	√	×	√	×	√
	`transform_if`	√	√	√	√	×	√	×	√
	`sequence`	√	√	√	√	×	√	×	√
	`tabulate`	√	√	√	√	×	√	×	√
Sort	`sort`	√	√	√	√	×[a]	√	√	√
	`sort_by_key`	√	√	√	√	×[b]	√	×	√
Scatter	`scatter`	√	√	√	√	×	√	×	√
	`scatter_if`	√	√	√	√	×	√	×	√
Gather	`gather`	√	√	√	√	×	√	×	√
	`gather_if`	√	√	√	√	×	√	×	√
Reduce	`reduce`	√	√	√	√	√	√	√	√
Transform reduce	`inner_product`	√	√	√	√	×	√	×	√

Table 1: Summary of results upon invoking a collection of `Thrust` functions. H-R means that the `Thrust` function was passed a raw host pointer and the iterator used was host-bound. H-W means that the `Thrust` function was passed a `Thrust` vector that wrapped a host pointer and that the iterator used was host-bound. H-M-R means that the `Thrust` function was passed a raw unified (managed) memory pointer and the iterator used was host-bound. H-M-W means that the `Thrust` function was passed a `Thrust` vector that wrapped a raw unified (managed) memory pointer and that the iterator used was host-bound. There is a set of similar columns in which H was replaced with D (from device).

[a] Sorting in ascending order (i.e., using the default comparator) produces the correct results; however, using a different comparator results in a segmentation fault.

[b] Sorting in ascending order (i.e., using the default comparator) produces the correct results; however, using a different comparator produces wrong results.

3 Discussion

One can notice that when unified memory pointers are wrapped in `Thrust` vectors, the function calls work fine both on the host and the device. If raw vectors are passed to `Thrust` calls, the device calls fail in almost all cases. Note that this is not an issue related to the use of Unified Memory. If anything, using a unified memory raw pointer in a device `sort` call works, while passing a raw device pointer to the same call doesn't. To conclude, the use of the Unified Memory in CUDA 6.0 works, at least in theory. To make it always work, one

has to wrap a unified memory pointer in a `Thrust` vector. This might add one line of code. Then, the same `Thrust` vector can be used for processing on either the host or the device side no matter whether the pointer wrapped is a unified memory pointer or not.

References

[1] Parallel Execution Policies. http://thrust.github.io/doc/group__execution__policies.html, 2013.

[2] Powerful New Features in CUDA 6. https://developer.nvidia.com/parallelforall/powerful-new-features-cuda-6, 2014.

[3] Thrust Library Introduction. https://developer.nvidia.com/thrust, 2014.

[4] Thrust Test. https://github.com/uwsbel/Thrust-test, 2014.

[5] Unified Memory in CUDA 6. http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, 2014.

Appendices

A Example use of `Thrust`'s `reduce` Function

The example shows how Unified Memory can be used with `Thrust`. We pick `reduce` as the sample code since in our experience this `Thrust` function has worked as expected under all scenarios.

```
const int ARRAY_SIZE = 1000;

enum Method {
  RAW,
  WRAPPED
};

bool reduce_test(Method method)
{
  double *mA;
  cudaMallocManaged(&mA, ARRAY_SIZE * sizeof(double));

  for (int i = 0; i < ARRAY_SIZE; i++)
    mA[i] = 1.0 * (i + 1);

  double maximum;

  switch (method) {
  case RAW:
    {
      maximum = thrust::reduce(thrust::cuda::par, mA, mA + ARRAY_SIZE, 0.0,
    thrust::maximum<double>());
      break;
    }
  case WRAPPED:
    {
      thrust::device_ptr<double> A_begin(mA), A_end(mA + ARRAY_SIZE);
      maximum = thrust::reduce(A_begin, A_end , 0.0, thrust::maximum<double>()
    );
      break;
    }
  default:
    break;
  }
  cudaDeviceSynchronize();

  bool result = (fabs(maximum - ARRAY_SIZE) < 1e-10);

  cudaFree(mA);

  return result;
```

```
}

int main(int argc, char **argv)
{
  std::cout << "Reduce DMR ... " << std::flush << reduce_test(RAW) << std::
    endl;
  std::cout << "Reduce DMW ... " << std::flush << reduce_test(WRAPPED) << std
    ::endl;
  return 0;
}
```

src/reduce.cu