Lab session 5: Streams and multiple GPUs

- **Solution to third problem from previous lab:**

```
struct ThickPlus {
  ThickPlus (int t) : thickness(t) {}

  __device__ int operator() (thrust::tuple<int, int*> t) {
    int ret = 0;
    int startIdx = thrust::get<0>(t)*thickness;
    for (int i = 0; i < thickness; ++i) {
      ret += thrust::get<1>(t)[startIdx + i];
    }
    return ret;
  }

  int thickness;
};

int main (int argc, char** argv) {
  int sizeOfVector = 100;
  if (argc > 1) sizeOfVector = atoi(argv[1]);
  int *hvector = new int[sizeOfVector];
  int *dvector;
  cudaMalloc((void**) &dvector, sizeOfVector*sizeof(int));
  srand(42);
```

```
  int checksum = 0;
  for (int i = 0; i < sizeOfVector; ++i) {
    hvector[i] = rand() % 10;
    checksum += hvector[i];
  }

  std::cout << "CPU sum: " << checksum << std::endl;

  cudaMemcpy(dvector, hvector,
             sizeOfVector*sizeof(int),
             cudaMemcpyHostToDevice);
  thrust::constant_iterator<int*> arrayPointer(dvector);
  thrust::counting_iterator<int> counter(0);

  ThickPlus mp(2);
  int sum = transform_reduce(make_zip_iterator(make_tuple(counter,
                                                            arrayPointer)),
                             make_zip_iterator(make_tuple(counter + 50,
                                                            arrayPointer)),
                             mp,
                             0,
                             thrust::plus<int>());

  std::cout << sum << std::endl;
}
```

- **Rewrite the good old dot-product calculation to use two streams. Divide the**

data into two or more chunks, then copy one chunk using `cudaMemcpyAsync` while processing another. Remember that you will need pinned memory on the host side. Try several different chunk sizes - for example, split the data into two, four, or eight chunks. Remember to use `cudaGetDeviceProperties` to check that our GPUs are in fact capable of asynchronous memory copying.

- **Once more, with feeling! Rewrite the dot-product example to use two GPUs. Use Thrust code or plain CUDA code as you like. To test this you probably want to request `ppn=2` from the Oakley batch system (ie one CPU per thread), and of course 2 GPUs!**

- **Some additional practice with Thrust iterators. Write a struct `PrintStruct` with an operator method taking a single integer argument. The operator should return 0, but should also use `cuPrintf` to print its argument. Now use a `counting_iterator` to launch a `transform` or `transform_reduce` (take your pick) with a `PrintStruct` as the unary operator.**

- **Extend your `PrintStruct` struct to have an additional operator method (yes, this is allowed!) which, instead of a single integer, takes a tuple of an integer and an integer pointer. This new operator should print out its integer argument and the corresponding element of the pointer. Use a `zip_iterator` of a `counting_iterator` and a `constant_iterator` to launch a `transform_reduce` with this second operator. Let the argument of the `constant_iterator` be a device-side array of integers which you fill beforehand with random numbers.**

- **Give your `PrintStruct` still a third operator method. This one should take a**

tuple of an int pointer and an integer. (Notice that the order of the arguments matters! Thrust will treat this as a separate operator from the previous one.) This operator should print two elements of the array, instead of one. Launch a `transform` or `transform_reduce` with enough threads that each element is printed once and only once. Be careful when you construct your `zip_iterator`; you want to be sure to target the correct operator.