HIGH PERFORMANCE GRAPH ANALYTICS 2021 CONTEST REPORT

Montiglio Luca
luca.montiglio@mail.polimi.it

ABSTRACT

The aim of the report is to illustrate the work done in the high-performance graph analytics 2021 contest.

***K*eywords** Personalized PageRank · GPU · large Graphs

1 Introduction

The assignment for the high-performance graph analytics 2021 contest was to implement the Personalized PageRank algorithm, more specifically to use the knowledge and the concepts from the course to parallelize part of the computation on GPU devices. The code would have been tested on the entire Wikipedia graph.

2 Execution

The code may be found at

`https://github.com/lm18-dev/-lm18-dev-high-performance-graph-analytics-2021-Montiglio-`

The choice was to adopt the spmv based implementation from the CPU algorithm, the code of which was provided to test the new code against, and to try to delegate to the GPU device its periodic computations, that were clearly easily parallelizable.[1]

The dot product between two vectors is a great example of a function that can be calculated faster on GPU. It's an already well studied and broadly used tool. For this reason, the use of the CUBLAS library dot product was preferred over an ad hoc kernel implementation.

```
personalized_pagerank.cu, Line: 267

cublasDdot(handle, V, x_dot_gpu, 1, pr_gpu, 1, dangling_factor_gpu);
```

A different approach was chosen for the euclidean distance function. The library compatibilities and the already chosen data type supported the decision to implement a new kernel. The implementation of the dot product function presented in CUDA By Example[2], already evaluated in the previous step as an alternative to the CUBLAS dot function, was modified to compute instead the euclidean distance as follows.

```
__global__ void gpu_euclidean_distance(const double *x, const double *y, const int N, double *
    result) {

  __shared__ float cache[128]; // 128 is the number of threads per block fixed in the kernel launch

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
  float temp = 0;
  while (tid < N) {
    temp += (x[tid] - y[tid])*(x[tid] - y[tid]);
    tid += blockDim.x * gridDim.x;
  }
  cache[cacheIndex] = temp;
  __syncthreads();
  int i = blockDim.x/2;
  while (i != 0) {
    if (cacheIndex < i){
      cache[cacheIndex] += cache[cacheIndex + i];
    }
    __syncthreads();
    i /= 2;
  }
  if (cacheIndex == 0)
  result[blockIdx.x] = std::sqrt(cache[0]);
}
```

```
personalized_pagerank.cu, Line: 320

gpu_euclidean_distance<<<num_blocks, 128>>>(pr_gpu, result_axpb_gpu, V, result_eudiff_gpu);
```

In a similar way, after understanding the similar needs in parallelism, the GPU axpb personalized kernel was written from a basic add vector kernel from the course slides adapted to the study case.

```
__global__ void gpu_axpb_personalized(double alpha, double *x, double beta,
const int personalization_vertex, double *result, const int N ) {

  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  double one_minus_alpha = 1 - alpha;

  while (tid < N) {
    result[tid] = alpha * x[tid] + beta + ((personalization_vertex == tid) ? one_minus_alpha :
        0.0);
    tid += blockDim.x * gridDim.x;
  }

}
```

```
personalized_pagerank.cu, Line: 287

gpu_axpb_personalized<<<num_blocks, block_size>>>(alpha, x_axpb_gpu, alpha * *dangling_factor_gpu
    / V, personalization_vertex, result_axpb_gpu, V);
```

3 Considerations

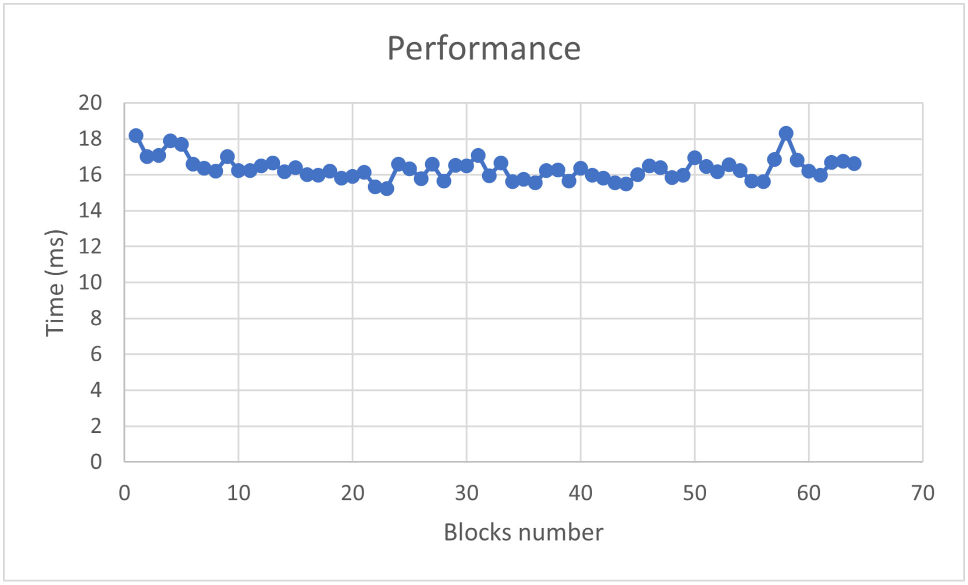
The overall implementation is far from optimal efficiency, although some considerations can be made for future use.

The main efficiency criticism can be found inside the loop: the need to reallocate on the GPU at each cycle vectors which are of the order of dimension of vertexes in the graph.

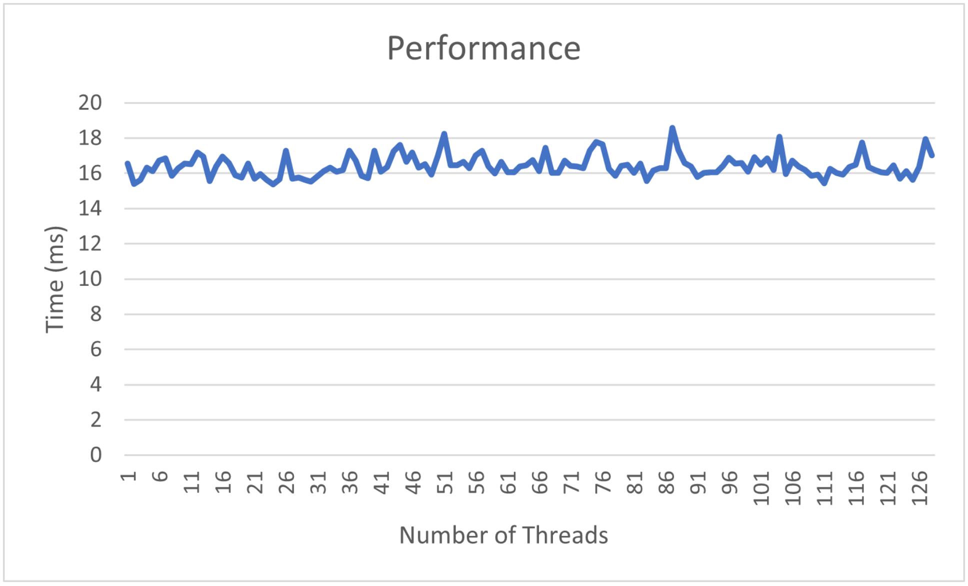
```
personalized_pagerank.cu, Line: 255

while (!converged && iter < max_iterations){

    ...

    cudaMemcpy( x_axpb_gpu, pr_tmp_gpu, V * sizeof(double),
        cudaMemcpyHostToDevice );

    ...

    cudaMemcpy( pr_tmp_gpu, result_axpb_gpu, V * sizeof(double),
        cudaMemcpyDeviceToHost );

    ...

    cudaMemcpy( err, result_eudiff_gpu, sizeof(double),
        cudaMemcpyDeviceToHost );

    ...

    cudaMemcpy( pr_gpu, pr_tmp_gpu, V * sizeof(double),
        cudaMemcpyHostToDevice );

    ...
```

A possible solution could be considering allocating the data on the RAM. A solution, although, that carries with it portability concerns and that was for this reason not embraced in the aim of the contest.

Finally, a brief experimental analysis was made to determine if something could be said on the number of blocks and threads in kernel launches. The former is the result of a simple python script, iterating the Personalized PageRank call with differents numbers of blocks in input in a range from 1 to 64, at a fixed number of threads: 128.



The latter is the output of the same script, with the number of blocks fixed at 64 and the number of threads in the range from 1 to 128.



The data doesn't show any significant correlation, and by that, the choice was to keep with the default number of blocks and threads provided.

Acknowledgments

Looking forward to further learning and researching possibilities, I would like to thank Alberto Parravicini, Francesco Sgherzi, Guido Walter Di Donato and the DEIB Politecnico di Milano for the great opportunity.

References

[1] Alberto Parravicini, Francesco Sgherzi, and Marco D. Santambrogio. A reduced-precision streaming spmv architecture for personalized pagerank on FPGA. *CoRR*, abs/2009.10443, 2020.

[2] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.