APUNTES, MANUALES, PRESENTACIONES

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Manual de desarrollo de algoritmo de extracción de envolvente convexa utilizando paralelización: massively parallel 3D gift-wrapping

Alfonso López Ruiz
Lidia María Ortega Alvarado

05/03/2025

Algoritmos geométricos

Lidia Ortega & Alfonso López
CS 3141: Prof. Lidia's Geometric Algorithms
March 5, 2025

Massively parallel 3D gift-wrapping

Goals. To build the 3D convex hull (CH) of a set of points, regardless of whether they belong to the vertices of a triangle mesh or a point cloud. Although there are many algorithms for extracting the CH of a point cloud, we will stick to an easy-to-understand algorithm such as Gift wrapping (also known as Jarvis March). It leverages simplicity by relying on fundamental geometric concepts. However, note that other algorithms are indeed much more efficient. Therefore, another goal of this project is to efficiently extract the CH with massively parallel programming with CUDA (in the Graphics Processing Unit), and OpenMP (in the Central Processing Unit).

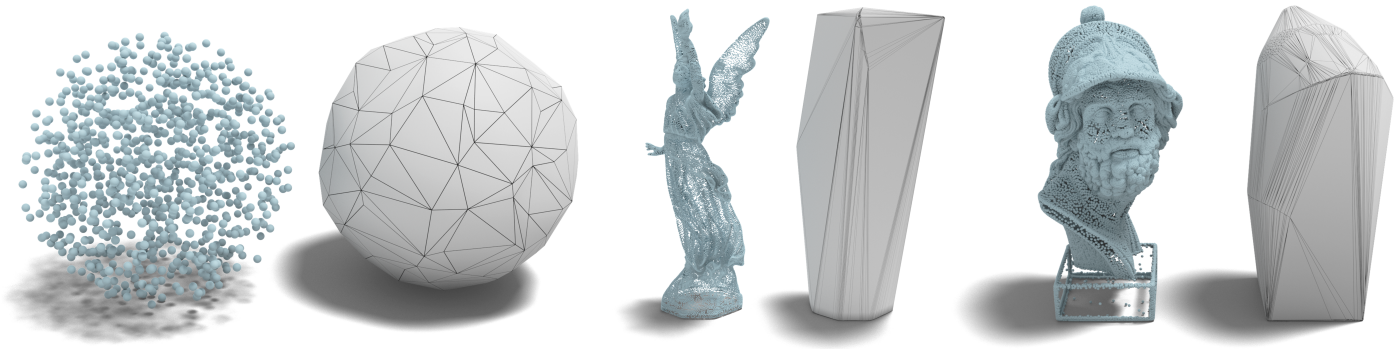


FIGURE 1. Examples of convex hulls extracted from random point clouds and triangle meshes.

Algorithm. The Jarvis march algorithm dates back to 1973 [2, 1], and although it was initially described for 2D point clouds, it can be trivially adapted to 3D. Instead of comparing segments with a point cloud, P, these segments can be turned into planes to check if they leave P on one end or another. Accordingly, a plane cannot be part of the convex hull, C, if part of P is left on one side and the rest on the other. Further considerations can be made to optimize the algorithm, such as organizing the point into quadrants, though we will stick to a base and easy-to-follow implementation.

Formally, a convex hull C of a point cloud P is the minimal polyhedron that can wrap P without holes. It is represented as if a band was stretched over P. Hence, it only touches points from P that belong to C.

More specifically, the fundamentals of Jarvis March are the following (Figure 2):

1) The point with minimum y coordinate, a, is part of C ($\mathcal{O}(n)$). The same applies to x, y and z.

2) If P is projected by discarding the z coordinate, a segment ab can be found since b is the point that has the lower angle w.r.t. a ($\mathcal{O}(n)$).

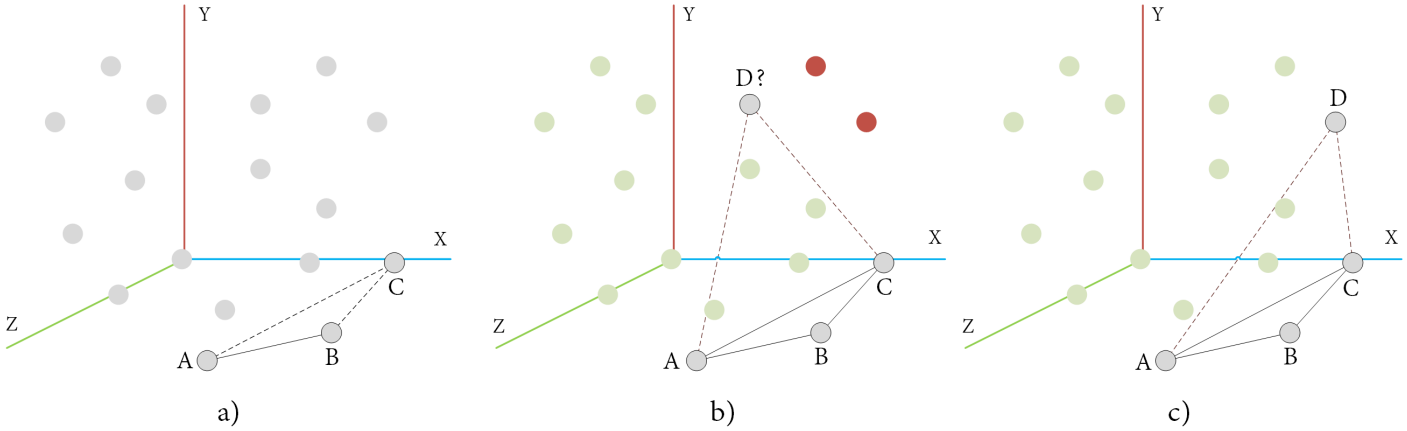


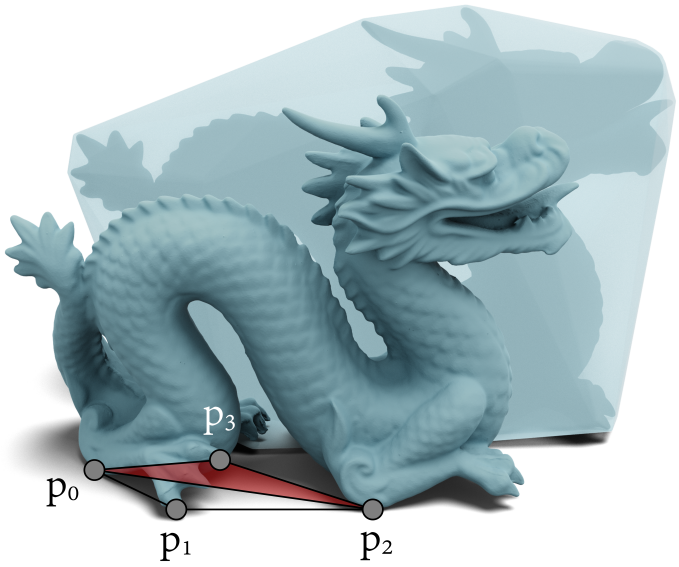
FIGURE 2. A possible pipeline for obtaining the first four points from the convex hull. In b), a point that does not meet the requirements is checked. Then, a valid point such as the one in c) is found. The red indicates the points above the plane, whereas the green indicates points below it.

3) A triangle abc can be obtained from a segment ab and a point c. It is part of C whether it leaves the rest of the points over the plane given by abc or on the same side ($\mathcal{O}(n)$).

4) The same procedure is followed for the rest of the triangles, using the previously found segments (for instance, ac, bc). Depending on the data structure storing these segments (stack or queue), the triangles of C can be found in a different order.

5) The following segments, p_ip_j, are extracted from this data structure, and when finding another valid point, p_k, the algorithm should omit the third vertex of the triangle where the segment was first observed. For a triangle abc and a segment bc, the algorithm should never check again a when such a segment is checked later in the process. Also, note that vertices may be shared by more than one triangle. Therefore, they cannot be discarded once found.

The following video showcases how to solve an iterative gift-wrapping algorithm using a stack and a queue.



Methodology. The procedure is defined below using a queue instead of a stack:

Data: Point cloud P with n points
Result: Convex hull as a set of triangles
Variables BoundaryCH: queue for retrieving edges, but it also should be able to remove any edge if necessary;
Variables PointsCH: maintains a list of points in the convex hull;
Project P into the XY plane, thus obtaining R;
Retrieve the point a with lowest y coordinate from R;
Retrieve a point b from R that composes a segment ab so that it leaves the rest of the points on one side;
Retrieve a point c from P such that the triangle abc leaves the rest of the points on one side;
BoundaryCH.insert(ab, bc, ca);
CH.insert(abc);
while *BoundaryCH not empty* **do**
  $de \leftarrow$ BoundaryCH.pop();
  Retrieve a point f from P such that the triangle def leaves the rest of the points on one side;
  CH.insert(def);
  if $F \notin$ *PointsCH* **then**
    PointsCH.insert(f);
    BoundaryCH.insert(dv, ev);
  end
  else
    if *BoundaryCH.contains(*df*)* **then**
      BoundaryCH.insert(df);
    end
    else
      BoundaryCH.remove(df);
    end
    if *BoundaryCH.contains(*ef*)* **then**
      BoundaryCH.insert(ef);
    end
    else
      BoundaryCH.remove(ef);
    end
  end
end

Algorithm 1: Pseudocode for 3D gift wrapping.

Resources. The following list of resources is provided as a starting point for solving this practice.

1) **Baseline renderer project (Github)**. As in the previous practices, the OpenGL project can be used as the baseline for rendering the point cloud and the resulting convex hull (composed of vertices and triangles). Though previously revised, the following piece of code refreshes how to define the geometry and topology (point cloud, wireframe, and triangle mesh) of any 3D model. Note that vertices can be flexibly defined by their position, normal vector, and texture coordinates. The latter two are not always required and thus can be omitted during definition. Similarly, not every topology is a must; for this project, only point clouds are necessary, whereas convex hulls will also be required to define their triangle mesh topology. From a triangle mesh, its wireframe and point cloud topology can be automatically extracted, and the same applies from wireframe to point cloud.

```
Component* component = new Component;

// Define geometry
component->_vertices.insert(component->_vertices.end(),
{
  VAO::Vertex { vec3(x, y, z), vec3(nx, ny, nz) },
  VAO::Vertex { vec3(x, y, z) },
  VAO::Vertex { vec3(x, y, z), vec3(nx, ny, nz), vec2(u, v) }
});

// Define topologies
component->_indices[VAO::IBO_TRIANGLES].insert(
component->_indices[VAO::IBO_TRIANGLES].end(),
{
  0, 1, 2,
  1, 2, 3
})

component->_indices[VAO::IBO_LINE].insert(
component->_indices[VAO::IBO_LINE].end(),
{
  0, 1
  1, 2
})

component->_indices[VAO::IBO_POINT].insert(
component->_indices[VAO::IBO_POINT].end(),
{
  0, 1, 2, 3, 4
})

// Automatic generation of other derived primitives
component->completeTopology();

// Send component data to GPU
this->buildVao(component);

// Add to vector to include in the rendering loop
this->_components.push_back(std::unique_ptr<Component>(component));
```

Listing 1. Definition of geometry and topology for a custom model.

2) CUDA handler as a higher-level abstraction to get access to reading, writing, error checks, and more in CUDA.

```
#pragma once

class CudaHandler
{
public:
  CudaHandler();
  virtual ~CudaHandler();

  static void checkError(cudaError_t result);

  template<typename T>
  static void downloadBufferGPU(T*& bufferPointer, T* buffer, size_t size);

  template<typename T>
  static void downloadBufferAsyncGPU(T*& bufferPointer, T* buffer, size_t
   size, cudaStream_t* stream);

  template<typename T>
  static void free(T*& bufferPointer);

  template<typename T>
  static void freeHost(T*& bufferPointer);

  static size_t getNumBlocks(size_t size, size_t blockThreads);

  template<typename T>
  static void initializeBufferGPU(T*& bufferPointer, size_t size, T* buffer
    = nullptr);

  template<typename T>
  static void initializeHostBufferGPU(T*& bufferPointer, size_t size, T*
   buffer = nullptr);

  static int setDevice(uint8_t deviceIndex = UINT8_MAX);

  static void startTimer(cudaEvent_t& startEvent, cudaEvent_t& stopEvent);

  static float stopTimer(cudaEvent_t& startEvent, cudaEvent_t& stopEvent);
};
```

Listing 2. Helpful class for using CUDA at a high-level abstraction.

3) Piece of CUDA code for identifying the **maximum and minimum floating point values** in a buffer, returning both the boundary value and the index on which such a value is stored. We provide this because CUDA natively supports the atomic operations for `int` types, whereas floating-point values require additional code.

```
...

__global__ void reduceMinIdxOptimized(const float* input, const int size,
    float* minOut, int* minIdxOut) {
    __shared__ float sharedMin;
    __shared__ int sharedMinIdx;

    if (0 == threadIdx.x) {
        sharedMin = FLT_MAX;
        sharedMinIdx = 0;
    }

    __syncthreads();

    float localMin = FLT_MAX;
    int localMinIdx = 0;

    for (int i = threadIdx.x; i < size; i += blockDim.x) {
        float val = input[i];

        if (localMin > val) {
            localMin = val;
            localMinIdx = i;
        }
    }

    const float warpMin = warpReduceMin(localMin);
    const int warpMinXY = warpBroadcast(localMinIdx, warpMin == localMin);
    const int lane = threadIdx.x % warpSize;

    if (lane == 0)
        fatomicMin(&sharedMin, warpMin);

    __syncthreads();

    if (lane == 0)
        if (sharedMin == warpMin)
            sharedMinIdx = warpMinXY;

    __syncthreads();

    if (0 == threadIdx.x) {
        *minOut = sharedMin;
        *minIdxOut = sharedMinIdx;
    }
}

...
```

Listing 3. Main function to extract the index (and value) of maximum argument in a buffer.

OpenMP. Open Multi-Processing, or OpenMP, is a multiplatform application development interface (API) for Unix and Windows O.S. that enables multi-threading programming using shared memory in coding languages such as C, C++ and Fortran. The main advantage of OpenMP over other multi-threading frameworks such as CUDA, OpenCL and OpenGL compute shaders is the ease of incorporating parallelism into our code (e.g., in `for` loops). Its main limitations come from the maximum number of simultaneous threads, given the CPU cores, whereas GPU-based platforms work over much more numerous and smaller cores.

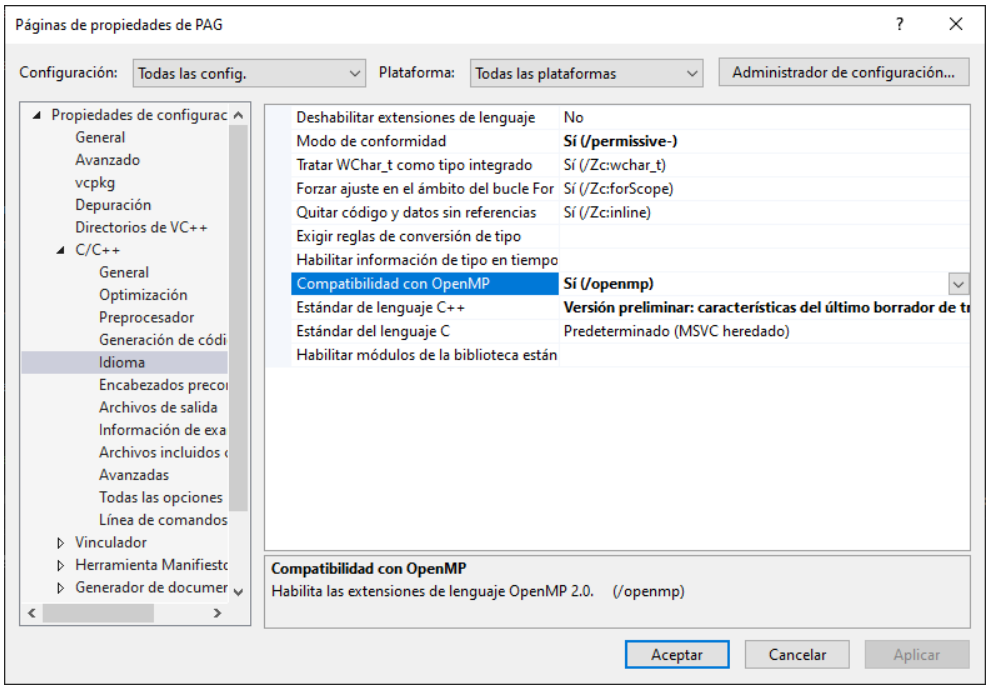


FIGURE 3. Enabling the OpenMP capabilities in a Microsoft Visual Studio project.

By default, Microsoft Visual Studio projects do not enable OpenMP, though it is easily enabled as shown in Figure 3. To this end, it is required to right-click on the project name and select its properties. Then, we can move to `C++ > Language` to modify OpenMP compatibility to `Yes (/openmp)`. **Warning**: Microsoft Visual Studio does not integrate the latest OpenMP version but provides a trivial way to incorporate it and use basic parallel definitions.

The OpenMP syntax is characterized by `#pragma omp directives`, which can be placed before a block of code that can be a `for` loop or a set of instructions delimited by {}. You can further check the instruction set in Microsoft's documentation, but some of the most important ones are the following:

- `#pragma omp parallel for` parallelizes a `for` loop so that each index i (`int`) will be handled by a different thread (at the same time if there are enough threads, or in successive bursts).
- `#pragma omp atomic`, for operations to be performed by only one thread at a time. It only affects one instruction: allocations, readings/writings, etc. In short, it affects simple operations such as `++i, x = v, v = x * y`...
- `#pragma omp critical`: block of code that can only be executed in one thread at a time. This block must be delimited with square brackets, except when it affects only one line.

- `#pragma omp barrier`: explicit synchronization of all threads at the point where this directive is included.
- `#pragma omp master / omp single`: executes a piece of code in a single thread, which can be the master or any of them (single).
- `private(var)`: creates a copy of an external variable for each thread to be safely modified. Alternatively, you can create variables within the code of each thread to avoid simultaneous access. The main difference with private is that the copied variable for each thread is still accessible after the multithreaded block ends.
- `omp_get_num_threads`: checks how many threads can be launched. It is different than `omp_get_max_threads` if the number of threads has been modified.
- `omp_get_thread_num`: returns the id of the operative thread.
- `omp_get_max_threads`: returns the number of threads that can work simultaneously at most.
- `omp_set_num_threads`: modifies the number of simultaneous threads.

Instead of letting OpenMP parallelize a `for` loop, we can manage the indices of a buffer that are handled by different threads. This choice is slightly more complex but enables reduction operations (e.g., min and max) without slowing the execution. Otherwise, `atomic` and `critical` blocks are traditionally performance bottlenecks that, if not addressed correctly, could even worsen the sequential performance. The following example shows how this can be done by dynamically adjusting it to the number of available threads:

```
1  std::vector<T> ompResults(omp_get_max_threads());
2
3  #pragma omp parallel
4  {
5    int threadCount = omp_get_num_threads();
6    int id = omp_get_thread_num();
7    int chunkSize = glm::ceil(data.size() / threadCount);
8    int start = id * chunkSize, end = glm::clamp((id + 1) * chunkSize, 0,
       static_cast<int>(data.size()));
9
10   for (int i = id * chunkSize; i < end; ++i)
11   {
12     // Thread core
13   }
14 }
15 {
16   // Sequential: merge thread results
17   for (auto& result : ompResults)
18   {
19     // Merge
20   }
21 }
```

Listing 4. A buffer operated in parallel by the maximum number of threads available.

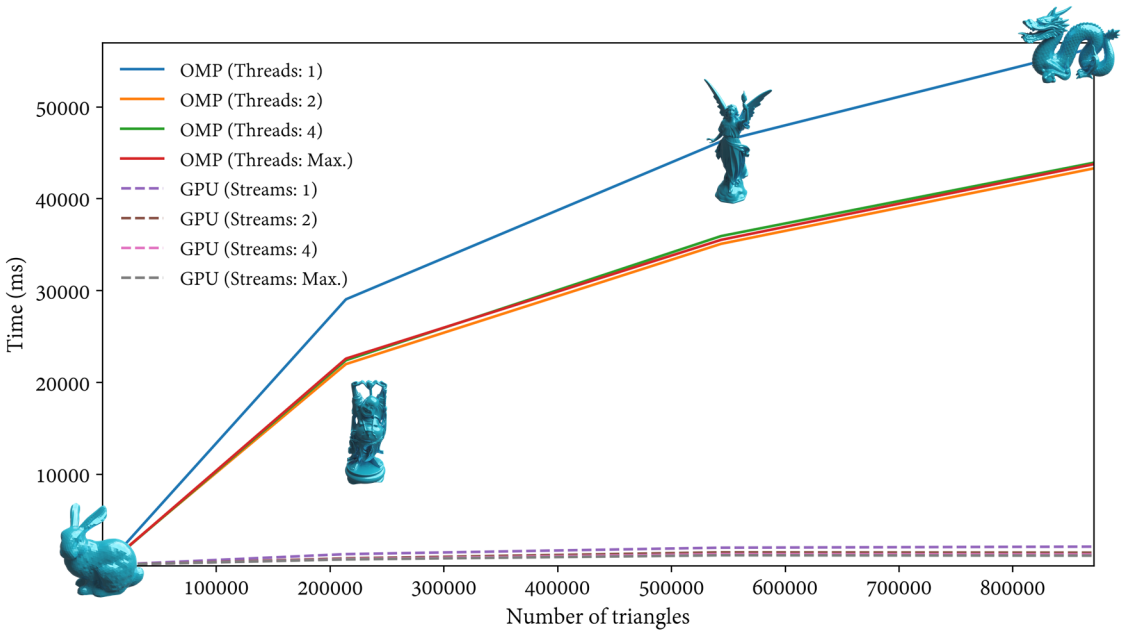


FIGURE 4. Comparison of response time in milliseconds of two gift-wrapping implementations using OpenMP and CUDA. The OpenMP version running with a single thread corresponds to the baseline sequential approach.

What is expected from you?

As shown in algorithm 1, the pipeline is mostly sequential; however each of its steps can be trivially parallelized by solving $\mathcal{O}(n)$ operations with several threads working simultaneously. Therefore, the previous code should solve almost every stage of the gift-wrapping algorithm. As depicted in Figure 4, there is a considerable speedup between a single thread and using several of them.

CUDA. The last effort in this project is to adapt our code to be executed using CUDA (Compute Unified Device Architecture) [3]. Besides being a GPU-based multi-threading framework, CUDA is also a GPU architecture released in 2007. In this manner, NVIDIA intended this new family of GPUs to be used for general-purpose computing. Unlike other frameworks such as OpenCL, NVIDIA only works on NVIDIA GPUs. Anyway, this should not be a great deal since NVIDIA currently has the ~80% of market shares.

Hence, CUDA provides a software layer that eases access to GPU resources. It is programmed similarly to C in compute kernels (typically labelled with `.cuh` and `.cu` extensions). In this regard, the CUDA suite also includes the compiler for these kernels, although IDEs such as Microsoft Visual Studio already include templates that get in charge of using the CUDA C compiler for kernel files.

First, it is required to install the CUDA toolkit (version 12.3 being the last as this is being written), which also includes the integration with MSVC IDE in the latest versions. Therefore, we will stick to it. What is left is to create an MSVC project following the CUDA template. Note that we can move our previous files to the new project or create a new CUDA-based project.

Compute kernels have a `.cu` extension (with `.cuh` for header files) and must be explicitly labeled in MSVC to ensure compilation with CUDA tools, as shown in Figure 5.

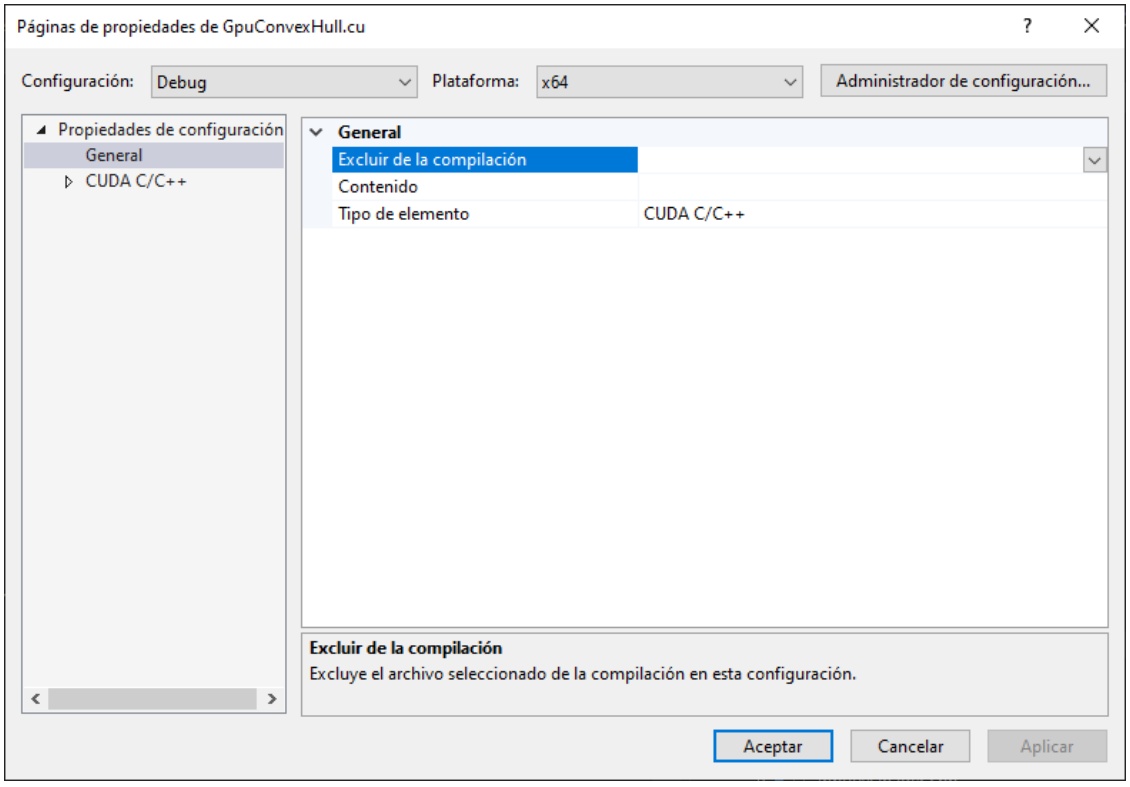


FIGURE 5. Compute kernel labeled as CUDA C file for its later compilation.

The intent of this project is not to dig deep into CUDA principles, and for this reason, a CUDA handler class is already provided. However, we highly encourage the reader to check the CUDA by example book [3] for obtaining a broad view of CUDA instructions. Although there are several ways in which gift-wrapping can be sorted out, we suggest readers solve this problem with code blocks such as 1) compute (and store) values and 2) extract minimum/maximum values. In this manner, the kernels

are significantly simplified and maximum/minimum operators can be reused. CUDA does not include native atomic operators for floating-point values; consequently, we have provided them as resources in this project.

The following piece of code summarizes the basics of CUDA regarding reading, writing, initializing buffers and calling kernels:

```
// Buffer initialization
CudaHandler::initializeBufferGPU(buffer, size, pointer);
CudaHandler::initializeBufferGPU(buffer, size);   // No data yet

// Transferring data to GPU
CudaHandler::checkError(cudaMemcpy(buffer, pointerOrigin, sizeof(T) * size,
    cudaMemcpyHostToDevice));

// Download data
memcpy(buffer, destinationPointer, sizeof(T) * size);

// Call to function
kernelName<<<CudaHandler::getNumBlocks(size, 1024), 1024, 0>>>(argument_1, ...,
    argument_n);
```

LISTING 5. Examples of basic CUDA instructions accessed through our CUDA interface.

Readers should notice that kernels receive three parameters in the angle brackets. The first is the number of thread blocks, and the second is the number of threads within each block, with the latter being more limited in size. For this reason, the number of threads is fixed to a reasonable number, and the number of blocks is scaled according to the buffer size and the number of threads within each block.

One example of a kernel function that can be implemented in our `.cu` file is the following:

```
__global__ void myFirstCudaKernel(const glm::vec3* points, const int size, ...,
    float* buffer)
{
    int threadID = threadIdx.x + blockIdx.x * blockDim.x;
    if (threadID < size)
    {
        buffer[threadID] = ...;
    }
}
```

LISTING 6. Example of a kernel that computes the thread ID and performs a writing operation over a buffer.

Finally, GPU utilization can be further optimized by parallelizing memory transfers and kernel execution. By allowing copy and compute engines to operate concurrently, idle time is minimized, improving overall performance. However, our gift-wrapping algorithm has few data transfers since it only requires querying maximum/minimum values. An alternative approach is to maximize GPU utilization by launching multiple parallel kernels, each solving several iterations concurrently. In any case, this parallelization is achieved by what is coined as streams. In this manner, asynchronous data transfer and kernels are linked to streams that work independently. Some examples of stream usage are the following:

```
// Create stream
std::vector<cudaStream_t> stream (numStreams);
CudaHandler::checkError(cudaStreamCreate(&stream[0]));

// Asynchronous data submission
CudaHandler::checkError(cudaMemcpyAsync(buffer, pointerOrigin, sizeof(T) * size,
    cudaMemcpyHostToDevice, stream));

// Kernel
cudaKernel<<<blocks, threadsBlock, 0, stream[0]>>> (...);

// Forced synchronization
CudaHandler::checkError(cudaStreamSynchronize(stream[0]));
```

LISTING 7. CUDA operations performed over a specific stream.

Response time measurement. Measurements for CPU-based processes are trivial to implement, whereas CUDA requires timers for it. The following piece of code shows how to trigger and stop timers to measure the response time:

```
cudaEvent_t start, stop;
CudaHandler::startTimer(start, stop);

std::cout << "Elapsed ms: " << CudaHandler::stopTimer(start, stop) << std::endl;

CudaHandler::checkError(cudaEventDestroy(start));
CudaHandler::checkError(cudaEventDestroy(stop));
```

LISTING 8. CUDA instructions for triggering and checking a timer.

REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, Cambridge, Massachusetts London, England, September 2009.

[2] Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, March 1973. Publisher: Elsevier.

[3] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Nvidia, Upper Saddle River, NJ Munich, July 2010.

CAMPUS LAS LAGUNILLAS, S/N, A3-102, JAÉN (23071)