# Project CUDC Report

OUNAN DING
Student ID: 861194909
oding001@ucr.edu

## Contents

**1 Overview** **1**

**2 Design and Implementation** **2**
2.1 General Overview . . . . . 2
2.2 The CPU Implementation in C++ . . . . . 4
2.3 The CUDA Implementation . . . . . 4

**3 Performance Test and Analysis** **5**
3.1 The CPU Implementation in C++ . . . . . 6
3.2 The CUDA Implementation . . . . . 6

**4 Summary** **7**

**References** **8**

# 1 Overview

This report is for the final project of CS/EE-217 GPU Architecture and Programming. Implementations of dual contouring will be presented and analyzed.

We provide a CPU implementation written in C++ first as our baseline, then move on to introduce the CUDA implementations. We will see what the bottle neck is in our CPU version, and how we make trade off in the CUDA version to overcome that bottle neck. A visualizer is also provided as a Blender add-on to preview the results and check the correctness.

Finally we will provide some performance testing and detailed analysis in the last part of this report.

# 2 Design and Implementation

## 2.1 General Overview

In this section we will provide a general overview of dual contouring. The architecture-independent perspectives of this algorithm will be discussed.

The primary paper used for this project is [JLSW02]. [SWU] also provides supplemental materials on how to construct and solve the Quadratic Error Function(QEF).

For the first step in dual contouring, given an implicit function such as $(x^2 + y^2 - 1)^3 - x^2y^3$ in 2D, or $x^2 + y^2 + z^2 - 1$ in 3D, we will sample the function on a uniform grid. Here is an example in figure 1, where we sample the function $x^2 + y^2 - 1.7$. The 0 level set of this function is plotted as a circle of radius 1.7. The samples which have a negative value are drawn with filled dots.

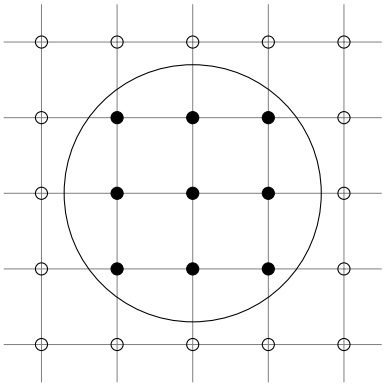

Figure 1: Sampling of $x^2 + y^2 - 1.7$

Every edge which exhibits opposite signs at its ends has a potential intersection with the 0 level set of the function. We highlight these edges in figure 2.

The intersections of these edges can be found by a binary search. And the gradients at the intersections (the Hermitian data in the glossary of [JLSW02]) can be computed by finite differencing. The result of this step is illustrated in figure 3.

After the Hermitian data is ready, the QEFs can be built as following:

$$\begin{aligned} E[x] &= \sum_i (n_i \cdot (x - p_i))^2 \\ &= x^T A^T A x - 2x^T A^T b + b^T b \end{aligned}$$

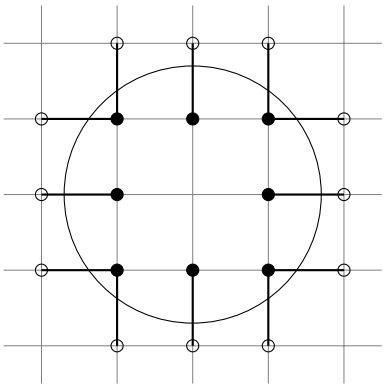

Figure 2: The intersected edges

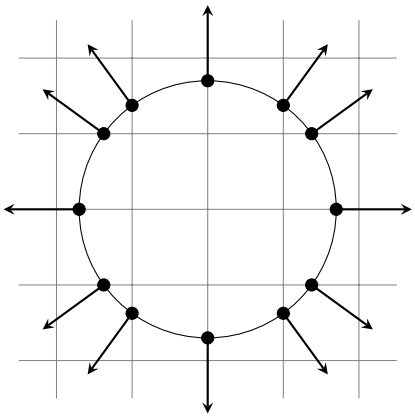

Figure 3: Intersections and their Hermitian data

In 2D dual contouring, each QEF can be represented by 7 floating point numbers. Since $A^TA$ is a $2 \times 2$ symmetrical matrix, it requires 3 floats. The $A^Tb$ is a $2 \times 1$ column vector, so it requires 2 floats. We also store 2 floats for the mass point, which is a $2 \times 1$ column vector and will be used in mass point projection (see [SWU] for details about mass point projection).

We solve the QEF by computing the pseudo-inverse as suggested in [JLSW02]. And compute a $2 \times 2$ SVD for the pseudo-inverse. This method is described in [Bli03].

After solving all of the QEFs, we get all vertices of the result mesh. We can build the topology by connecting vertices from adjacent grid cells.

Figure 4 illustrates dual contouring results of a heart, a squircle($x^4 + y^4 - 1$), and the subtraction of them.

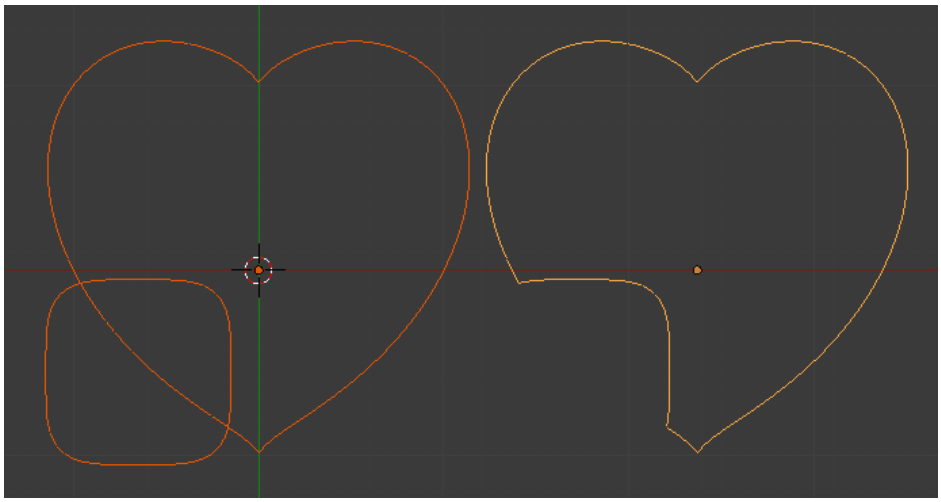

Figure 4: 2D dual contouring example

## 2.2 The CPU Implementation in C++

There are several considerations in our implementation of dual contouring on CPU.

First, a trivial implementation which stores the whole mesh grid suggests a spatial complexity of $\mathcal{O}(n^2)$(or $\mathcal{O}(n^3)$ in 3D). We do not want the memory consumption be a bottle neck, so we do the sampling in a row by row manner. Then only one row has to be maintained during sampling, and one extra row has to be maintained for the topology construction since we have to connect edges to the vertices in the previous row. Thus the spatial complexity can be reduced to $\mathcal{O}(n)$ in 2D(or $\mathcal{O}(n^2)$ in 3D).

Second, we think the sampling would occupy the majority time of the whole computation(However this turns out to be *false*. See our later performance report and analysis). So we want to reduce the computation as much as we can. Observing that each sampling vertex has 4 adjacent grid cells, we can do the sampling only once, cache it in memory, and reuse it for all adjacent grid cells.

The computing scheme is illustrated in figure 5. Each dots indicate a sampling point. And the dashed rectangle indicates the row we are working on. The hatched area indicates a 4 elements wide SIMD pack if SSE is used.

## 2.3 The CUDA Implementation

Being similar to the CPU implementation, we also use the row by row approach in the CUDA version to reduce the spatial complexity. And to utilize

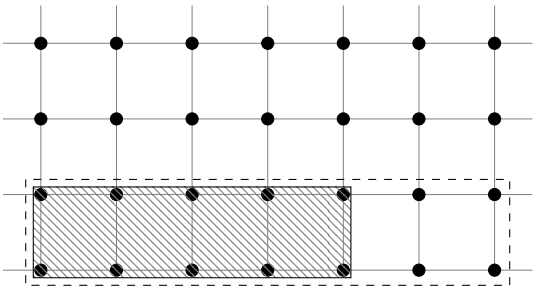

Figure 5: The computing scheme in C++

the high throughput of GPU we expand each row into a tile.

In figure 6 the computing scheme is illustrated. The dashed bounding rectangle indicates a CUDA grid; one CUDA block is drawn as solid rectangle; and one CUDA thread is drawn as hatched rectangle.

Should we cache the sampled values like what we did in CPU implementation? A preliminary performance test in CPU implementation reports that the sampling occupies only 5% of the whole algorithm, while the memory access occupies almost rest of the running time. Therefore we decide not to cache sampled values. That is, we sacrifice some duplicated computations for less memory access.

Concretely, every grid cell(which is mapped to a CUDA thread) computes 4 samples at its corner. Then every grid vertex gets sampled for 4 times by its adjacent cells. In this way we can avoid the writing to the memory.

Besides, since not every grid cell have intersections and contribute one QEF, we have to do a streaming compaction. In a traditional implementation of streaming compaction, prefix sum is implemented and an extra global array will be filled to do prefix sum across blocks. In our implementation, we use `__syncthreads_count` to count the sum in one block, and use atomic operations to compute the offset of one block. Hence no global array is required.

# 3 Performance Test and Analysis

All performance tests are taken on Windows 10 x86_64, with Visual C++ 2013 and CUDA toolkit 7.5. The CPU is i5 6600k @ 4.17GHz with 16 GB DDR4 RAM; the GPU is GTX 970.

The test case is to dual contour a heart equation on the domain $[-500, 500] \times [-500, 500]$ with grid resolution of 20000.

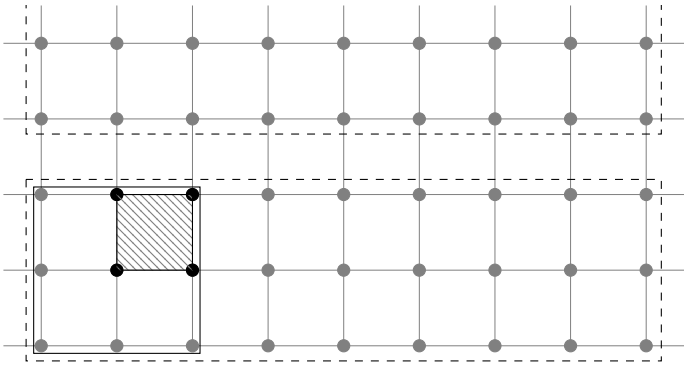

Figure 6: The computing scheme in CUDA

## 3.1 The CPU Implementation in C++

We use Intel's Amplifier XE to analyze the detailed time consumption of our CPU implementation. The total running time is 3.136 seconds. The top-down tree of the running time is shown in table 1.

| Function Stack | CPU Time: Total | CPU Time: Self |
|---|---|---|
| `CollectIntersectionEdge2` | 40.4% | 1.232s |
| `ConstructQEF2` | 40.1% | 1.222s |
| `GenericDualContour2` | 12.0% | 0.367s |
| `Sample2` | 6.7% | 0.203s |

Table 1: Top-down tree of the running time of CPU implementation

The bottle neck is in the gathering of intersected edges and QEF construction, which is also a gathering operation.

By inspecting the hot spot in the source code, we can find that about half of the time is in the reading of the memory. Therefore our implementation is memory bound.

Because the sampling only occupies a minority of runtime, we can trade some computation for better memory performance.

## 3.2 The CUDA Implementation

The total running time is 0.556 seconds. It is about 5.64x speed-up comparing with the CPU implementation.

We use nVidia's Visual Profiler to give detailed information of running time. It is shown in table 2.

Since most of the cells do not contribute to the zero level set of the function, so we can see the divergence does not cause problem.

| | KDualContour2 | KBuildTopology |
|---|---|---|
| Global Memory Load Efficiency | 0% | 82.1% |
| Global Memory Store Efficiency | 82.1% | 0% |
| Shared Memory Efficiency | 23.4% | 23.4% |
| Branch Efficiency | 100% | 100% |

Table 2: Details of the running time of CUDA implementation

The kernel `KDualContour2` does not read from the global memory, and `KBuildTopology` almost does not store to the global memory, therefore we get a 0% in the statistics.

We also see a 82.1% global memory load efficiency for `KDualContour2` and 82.1% global memory store efficiency for `KBuildTopology`. This memory access is to store and load the QEF indices. Most threads either write to a consecutive memory area or never write anything. That is, it is coalesced.

In the profiler we can see many kernel invocations. This is because we launch the kernels `KDualContour2` and `KBuildTopology` in an interleaved manner. That is, we solve QEFs for one row and construct topology for that row, then move on to the next row and repeat the same procedure. Too many kernel invocation from host may be a problem. But it requires further inspection.

# 4 Summary

# References

[Bli03] J. Blinn. *Jim Blinn's Corner: Notation, Notation, Notation*. Jim Blinn's corner. Morgan Kaufmann Publishers, 2003.

[JLSW02] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. *ACM Transactions on Graphics (TOG)*, 21(3):339–346, 2002.

[SWU] Scott Schaefer, Joe Warren, and Rice UniversityE. Dual contouring: The secret saucec.