OpenMP, сравнение GPU с CPU, как профилировать, как оптимизировать

Вычисления на видеокартах. Лекция 11

1. OpenMP
2. CPU и GPU профилировщики
3. Как оптимизировать

Полярный Николай

polarnick239@gmail.com

OpenMP

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    // код

    #pragma omp critical
    {
        // критическая секция
    }
}

#pragma omp parallel for schedule(static)
for (int i = 0; i < n; ++i) {
    // каждый поток назначен на фиксированные индексы (static - по умолчанию)
    // удобно для NUMA (с точки зрения доступа к памяти)
}

#pragma omp parallel for schedule(dynamic, 100)
for (int i = 0; i < n; ++i) {
    // каждый поток берет очередную группу индексов как только обработал предыдущую
}
```

CPU профилировщики

- Intel VTune Amplifier (Free 90-Day Renewable Community License)
- AMD CodeXL
- AMD μProf
- Perf + FlameGraph

GPGPU профилировщики

- AMD CodeXL
- NVIDIA Visual Profiler

Профилировщики общего назначения

- Shotgun профилирование остановкой под отладчиком
- Логгирование суммарного времени выполнения кернелов, времени работы всего алгоритма, времени работы CPU-части, времени IO, и т.о. прикладывать усилия по оптимизации в **узком** месте
- Посмотреть на графики утилизации CPU/GPU/IO/network и т.о. определить со стороны во что упирается
- Запуститься на одном/двух/четырех/64 ядрах и замерить ускорение
- Запуститься на одной/двух/четырех/восьми видеокартах и замерить

Как оптимизировать

1) Найти типичный сценарий выполнения и типичное железо
2) Запустить и найти узкое место (профилировщиками)
3) Оценить какой общий максимальный прирост в теории можно получить (в предположении что старое узкое место станет идеально быстрым)
4) Осознать можно ли его ускорить
5) Соотнести затраты разработки к выигрышу и выбрать решение:

- **Упорное** - начать с оптимизаций с наилучшим соотношением теоретического ускорения к времени разработки, постоянно убеждаться что общее время уменьшилось так как ожидалось, когда оставшееся теоретическое ускорение становится слишком сложным - **не забыть остановиться**
- **Административное** - дожать железом поставив более эффективную железку ответственную за узкое место (может быть выгоднее купить процессор с меньшим числом ядер, но с большим)
- **Философское** - не оптимизировать

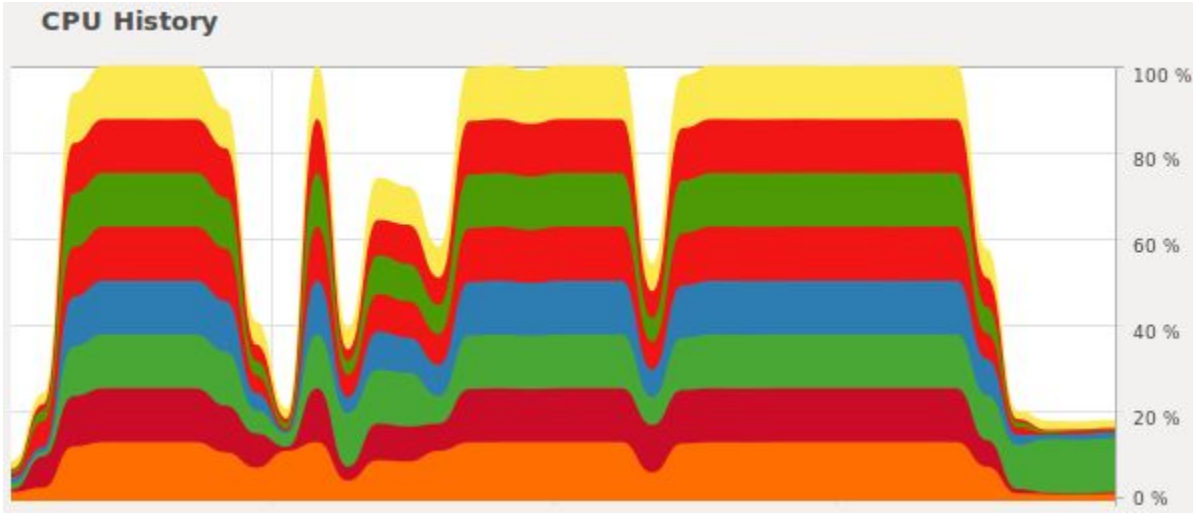
mkazhdan/PoissonRecon

CPU: i7 6700 (4 cores/8 threads)

Флаги компиляции: -O3 -funroll-loops -ffast-math

Аргументы запуска: Bin/Linux/PoissonRecon --in eagle.points.ply --out eagle.screened.color.ply --depth 10 --colors --performance --threads 8

График нагрузки на процессор:



Data-Parallel Octrees for Surface Reconstruction, Zhou et al., 2011

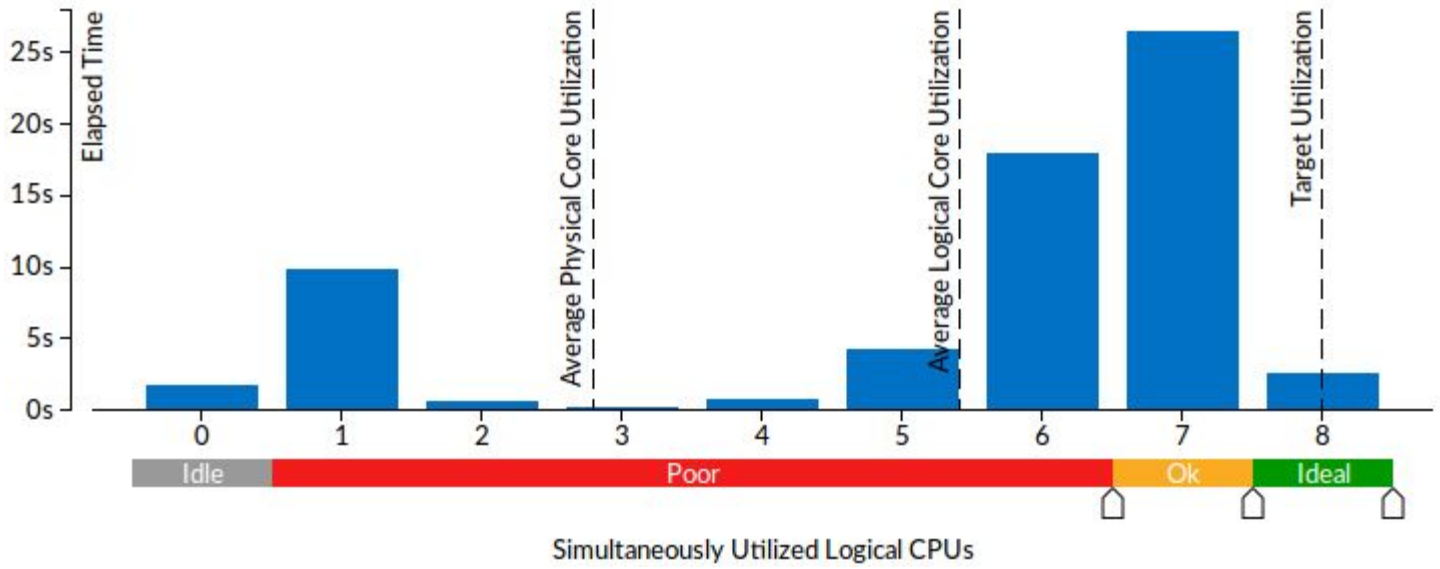
GeForce 8800 ULTRA быстрее референсной реализации в **100-200 раз**:

Model	# Points	Tree Depth	# Triangles	Mem_{ot}	Mem	T_{ot}	T_{func}	T_{iso}	T_{total}	FPS	T^{ot}_{cpu}	T_{cpu}
Bunny	353272	8	228653	120MB	290MB	40ms	144ms	6ms	190ms	5.26	8.5s	39s
Buddha	640735	8	242799	160MB	320MB	50ms	167ms	35ms	252ms	3.97	16.1s	38s
Armadillo	512802	8	201340	140MB	288MB	43ms	149ms	5ms	197ms	5.06	12.8s	42s
Elephant	216643	8	142197	200MB	391MB	46ms	209ms	41ms	296ms	3.38	5.5s	34s
Hand	259560	8	184747	125MB	253MB	36ms	143ms	27ms	206ms	4.85	6.4s	26s
Dragon	1565886	9	383985	230MB	460MB	251ms	486ms	23ms	760ms	1.31	39.1s	103s

In terms of performance, the GPU algorithm is over two orders of magnitude faster than the CPU algorithm. For example, for the Stanford Bunny, the GPU algorithm runs at 5.2 frames per second, whereas the CPU algorithm takes 39 seconds for a single frame. Note that the CPU implementation is provided by the authors of [2] and is well optimized. **(на самом деле нет)**

Intel VTune Amplifier

Профилировщик тоже показывает хороший уровень параллелизма:



Тем не менее PoissonRecon в зависимости от числа потоков исполняется:

- 1 поток - **60** секунд
- 2 потока - **43** секунды
- 8 потоков - **38** секунд

1 поток

Elapsed Time: 60.593s

SP GFLOPS: 0.357

Effective Physical Core Utilization: 23.4% (0.935 out of 4)

- Effective Logical Core Utilization: 12.2% (0.975 out of 8)
- **Effective CPU Utilization Histogram**

Memory Bound: 14.0% of Pipeline Slots

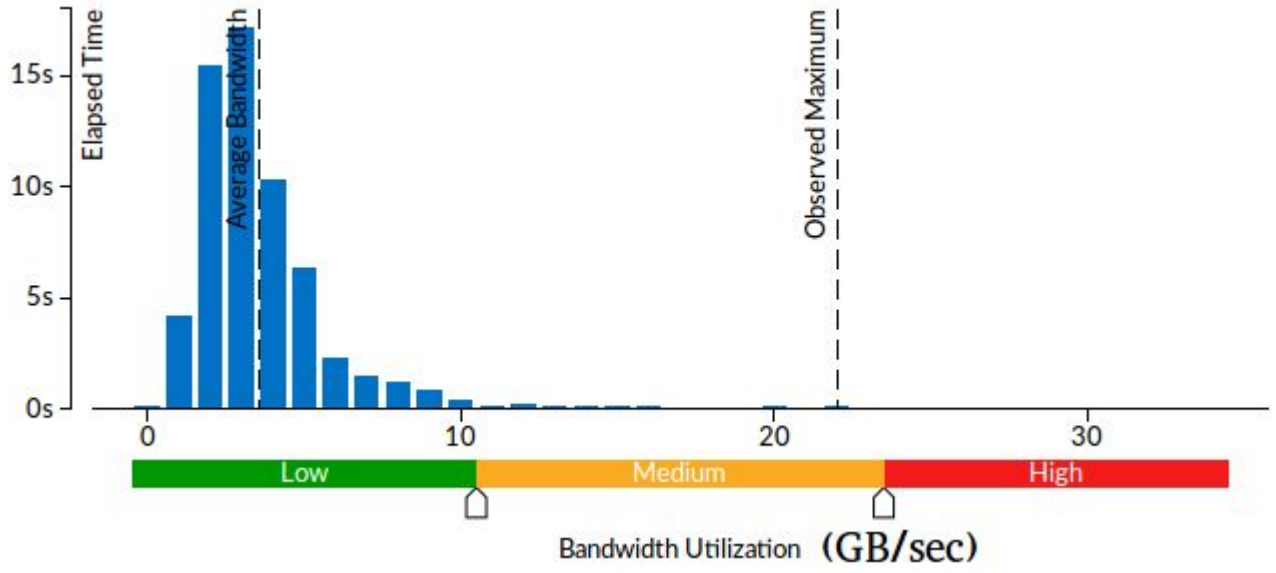
- Cache Bound: 8.5% of Clockticks
- **DRAM Bound**: 12.2% of Clockticks
- **Bandwidth Utilization Histogram**

FPU Utilization: 0.3%

- SP FLOPs per Cycle: 0.109 Out of 32
- Vector Capacity Usage: 13.8%
- FP Instruction Mix:
  - % of Packed FP Instr.: 0.0%
    - % of 128-bit: 0.0%
    - % of 256-bit: 0.0%
  - % of Scalar FP Instr.: 100.0%
- FP Arith/Mem Rd Instr. Ratio: 0.159
- FP Arith/Mem Wr Instr. Ratio: 0.461

1 поток

- Низкая утилизация ALU (0.357 гигафлопс)
- Упирается в оперативную память, низкая пропускная способность:



Многопоточный вариант должен бы успевать запрашивать большее количество данных?

8 потоков

Elapsed Time: 63.119s

SP GFLOPS: 0.335

Effective Physical Core Utilization: 79.2% (3.170 out of 4)

- Effective Logical Core Utilization: 76.9% (6.154 out of 8)
- **Effective CPU Utilization Histogram**

Memory Bound: 52.3% of Pipeline Slots

- Cache Bound: 29.1% of Clockticks
- **DRAM Bound**: 2.9% of Clockticks
- **Bandwidth Utilization Histogram**

- Memory Bound: 48.5%
  - L1 Bound: 20.6%
  - L2 Bound: 0.6%
  - L3 Bound: 6.4%
  - DRAM Bound: 3.0%

FPU Utilization: 0.1%

- SP FLOPs per Cycle: 0.022 Out of 32
- Vector Capacity Usage: 13.7%
- FP Instruction Mix:
  - % of Packed FP Instr.: 0.0%
    - % of 128-bit: 0.0%
    - % of 256-bit: 0.0%
  - % of Scalar FP Instr.: 100.0%
- FP Arith/Mem Rd Instr. Ratio: 0.146
- FP Arith/Mem Wr Instr. Ratio: 0.433

CUDA on Turing Opens New GPU Compute Possibilities

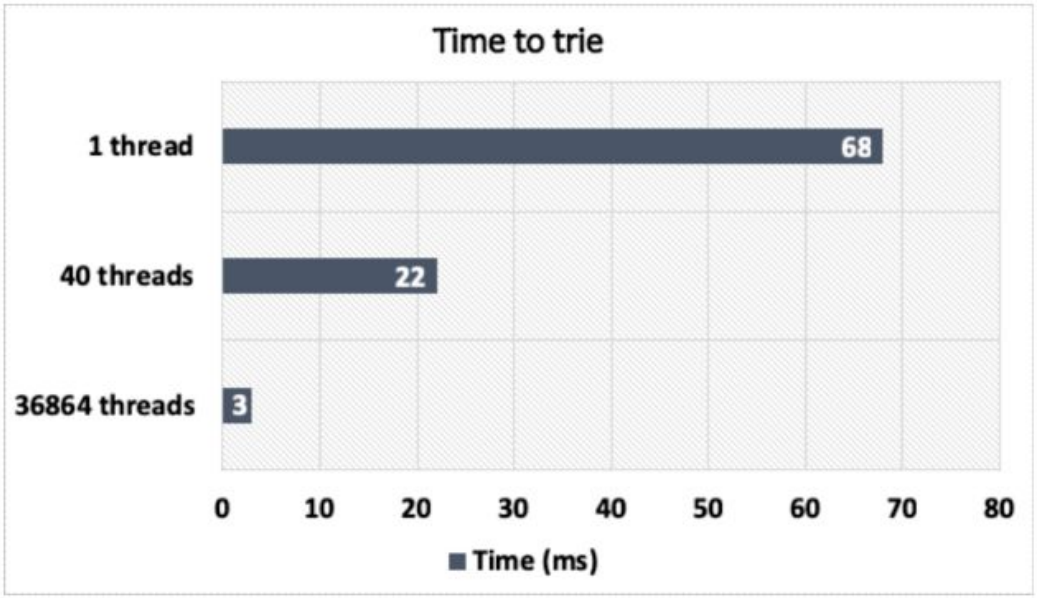


Figure 4. Dual Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz (40 cores total); GeForce RTX 2070 Founders Edition, Driver 410.73; average of cold and hot run

CppCon 2018: Olivier Giroux “High-Radix Concurrent C++”

```
[ogiroux@dt03 samples]$ ./trie_mt
Assembled 98881 nodes on 1 cpu threads in 127ms.
Assembled 98881 nodes on 1 cpu threads in 89ms.
Assembled 98881 nodes on 40 cpu threads in 14ms.
Assembled 98881 nodes on 40 cpu threads in 13ms.
```

Xeon с 40 ядрами по 3.00GHz, любопытно что в публикации другие числа

```
ngvpn01-172-29:samples olivier$ ./trie_mt
Assembled 98881 nodes on 1 cpu threads in 61ms.
Assembled 98881 nodes on 1 cpu threads in 68ms.
Assembled 98881 nodes on 8 cpu threads in 14ms.
Assembled 98881 nodes on 8 cpu threads in 15ms.
```

Ноутбук, вероятно с 4 ядрами

Оптимизации

1) Бывает выгодно пока видеокарта считает уже начинать готовить входные данные на CPU для следующего вычисления (**interleaving**)
2) Бывает выгодно запускать вычисления на одной и той же видеокарте в несколько потоков (параллелизм на уровне задач - ради **interleaving**)
3) Бывает выгодно перенести больше кода на видеокарту, т.к. их может быть много штук и тогда CPU может начать не успевать их кормить
4) Использовать процессор для вычислений при использовании множественных видеокарт - плохая идея, т.к. он и так еле успевает их кормить
5) Бывает выгодно при маленьких объемах задач - обрабатывать на CPU
6) Бывает выгодно увеличивать объем задачи, чтобы не было больших удельных накладных расходов на запуск, но нужно не забывать про driver timeout
7) Может быть выгоден шедулер для прогрузки общих для всех видеокарт данных, если IO занимает много времени