Semester Project Presentation

Alexandre Carlessi

Sahand Kashani-Akhavan

Implementing generic multiple-precision arithmetic on CUDA GPUs

Why GPU?

- Traditionally used for graphics
- Specialized in parallel processing of independent data
- Computational throughput has increased rapidly
- Scientists are now starting to use them for high performance computations (simulations, research, …)

What is CUDA?

- Framework made by NVIDIA to promote use of GPUs for computations
- C++ extension
- Allows programmer to access GPU memory
- Can create custom computational kernels

Our Project

- Learning GPU programming through experimentation to see what works best
- Experiment: multiple-precision arithmetic

Goals

- Wanted implementation to be
  - Re-usable
  - Have configurable precision
  - Efficient
  - No need of user knowledge about internals (like a library)

First steps

- Started implementing everything in C
- Advantages:
  - Pre-processor can make custom code
- Downside:
  - Not enough low-level control (e.g. cannot access carry bits easily)
  - Hard to avoid thread divergence and optimize simultaneously
  - Pre-processor can only create one version of the optimized code

Initial Solution

- Assembly
- Advantages
  - Good low-level control
  - Not hidden behind a compiler: we know exactly what's going on
- Downsides
  - Hard to debug
  - Like C, only one version of optimized code

Chosen Solution

- Meta-programming using scripts.
  - Use scripts to generate precision-specific optimized assembly code (handles corner cases)
- Advantages?
  - Power of assembly
  - Configurable output
  - Straight-line unrolled assembly
  - Multiple versions of optimized code

Our Framework

constants.py

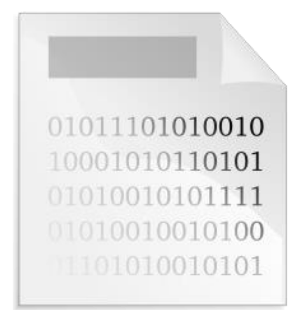
- Precision
- Bits per word
- Output file
- Block & thread count
- …

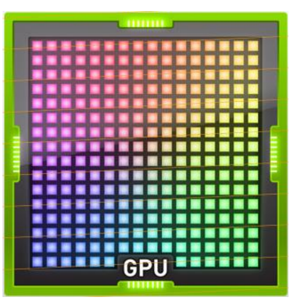
operator_generator.py

```
#define add_loc(c_loc, a_loc, b_loc)\
{\
    asm("add.cc.u32  %0, %1, %2;" : "=r"(c_loc[0]) : "r"(a_loc[0]), "r"(b_loc[0]));\
    asm("addc.cc.u32 %0, %1, %2;" : "=r"(c_loc[1]) : "r"(a_loc[1]), "r"(b_loc[1]));\
    asm("addc.cc.u32 %0, %1, %2;" : "=r"(c_loc[2]) : "r"(a_loc[2]), "r"(b_loc[2]));\
    asm("addc.cc.u32 %0, %1, %2;" : "=r"(c_loc[3]) : "r"(a_loc[3]), "r"(b_loc[3]));\
    asm("addc.u32    %0, %1, %2;" : "=r"(c_loc[4]) : "r"(a_loc[4]), "r"(b_loc[4]));\
}
```

operations.h

nvcc -arch=sm_30 file.cu





execution

Implementation

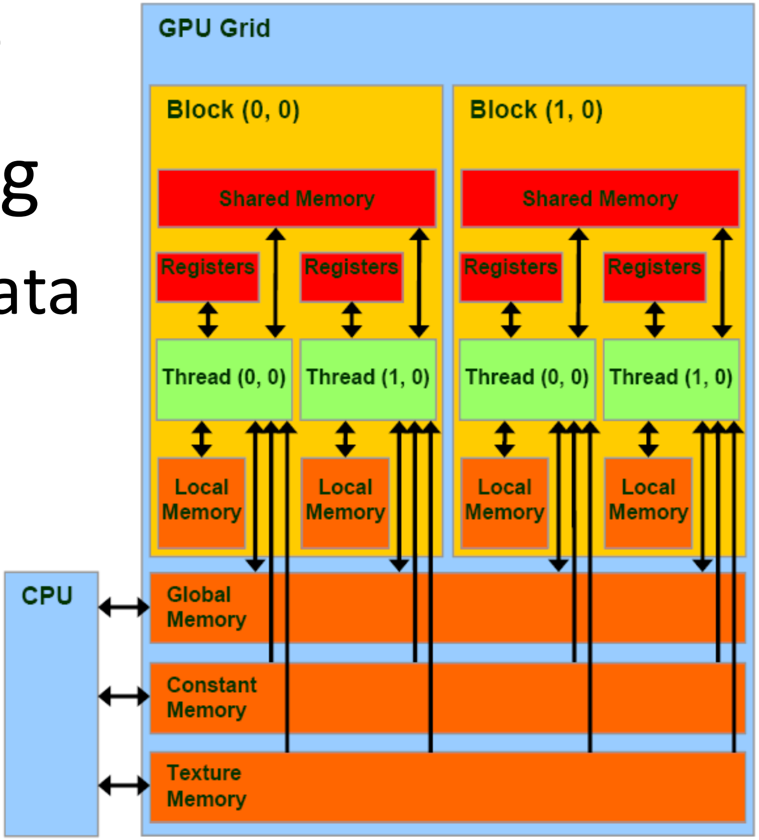
- Number representation
  - Unsigned integer array
  - Uses primary data type of device for efficiency
  - Two’s complement representation

a[0]	a[1]	a[2]	a[3]	a[4]
0xB65B9045	0x5E2503DD	0x372139F3	0xADBFB00E	0x00000004

a[0]	a[1]	a[2]	a[3]	a[4]
0xA93D318A	0x48EB68C1	0x7FB3AA74	0x9788816E	0xFFFFFFFE

Operation Memory Access

- 2 versions of each operator
- Differ in operand addressing
  - "global" operator: expects data to be in global coalesced memory
  - "local" operator: expects data to be in local non-coalesced memory

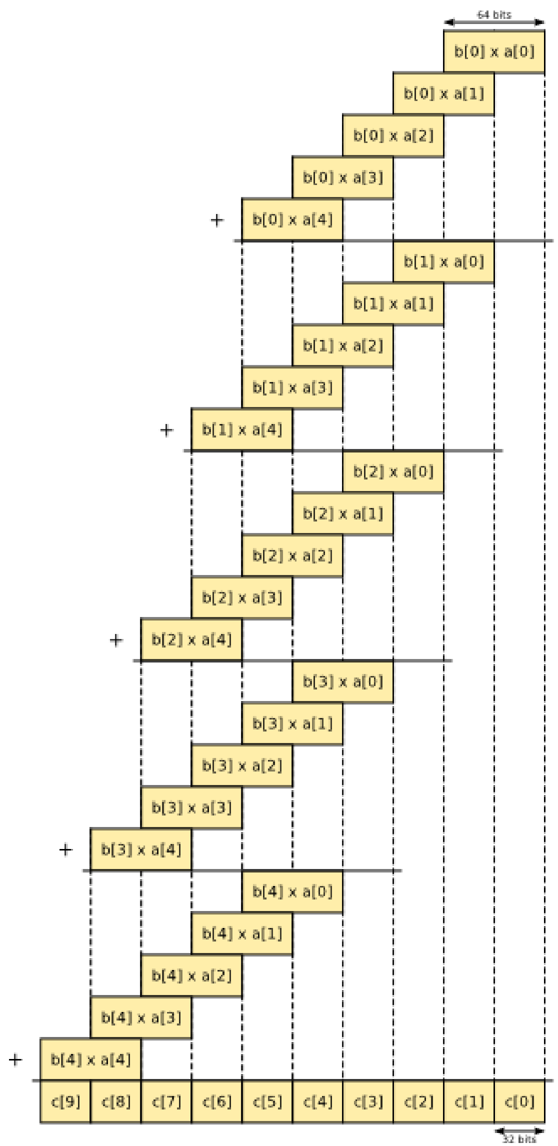


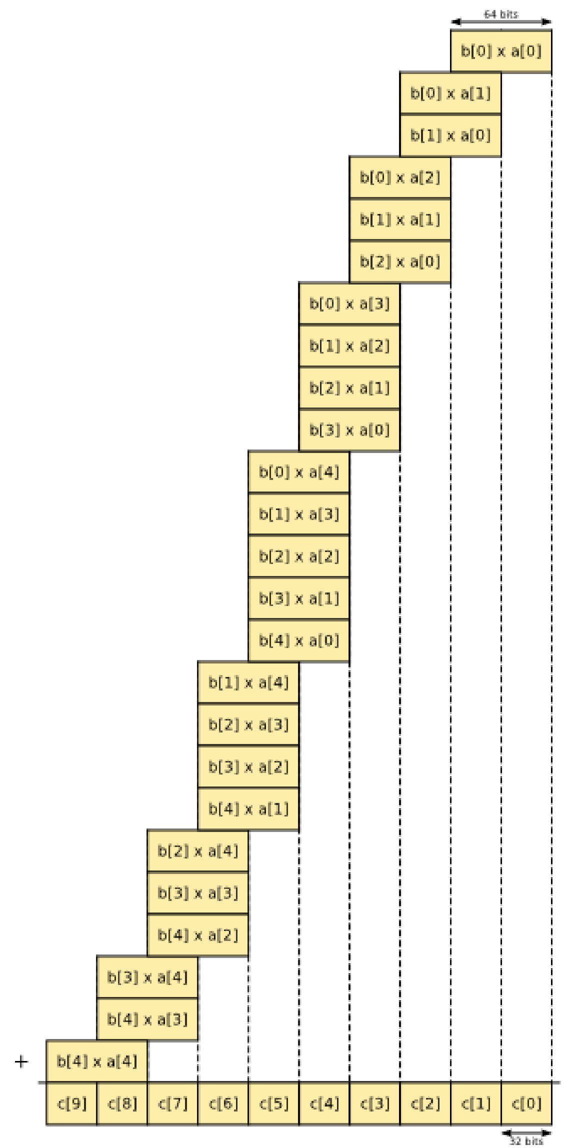
Implemented Operations

- Addition
- Subtraction
- Multiplication
- Karatsuba Multiplication (1-level)
- Modular Addition
- Modular Subtraction
- Montgomery Reduction*

* unfinished

Multiplication





Benchmark Infrastructure

->

compile

Executable

GMP

Generate coalesced bignum arrays

nvprof

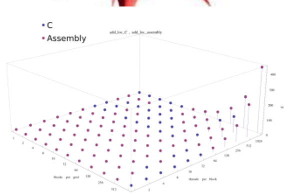
nvprof --print-gpu-trace --normalized-time-unit us executable

Benchmark results

DATA

mathematica

<-



Benchmarks

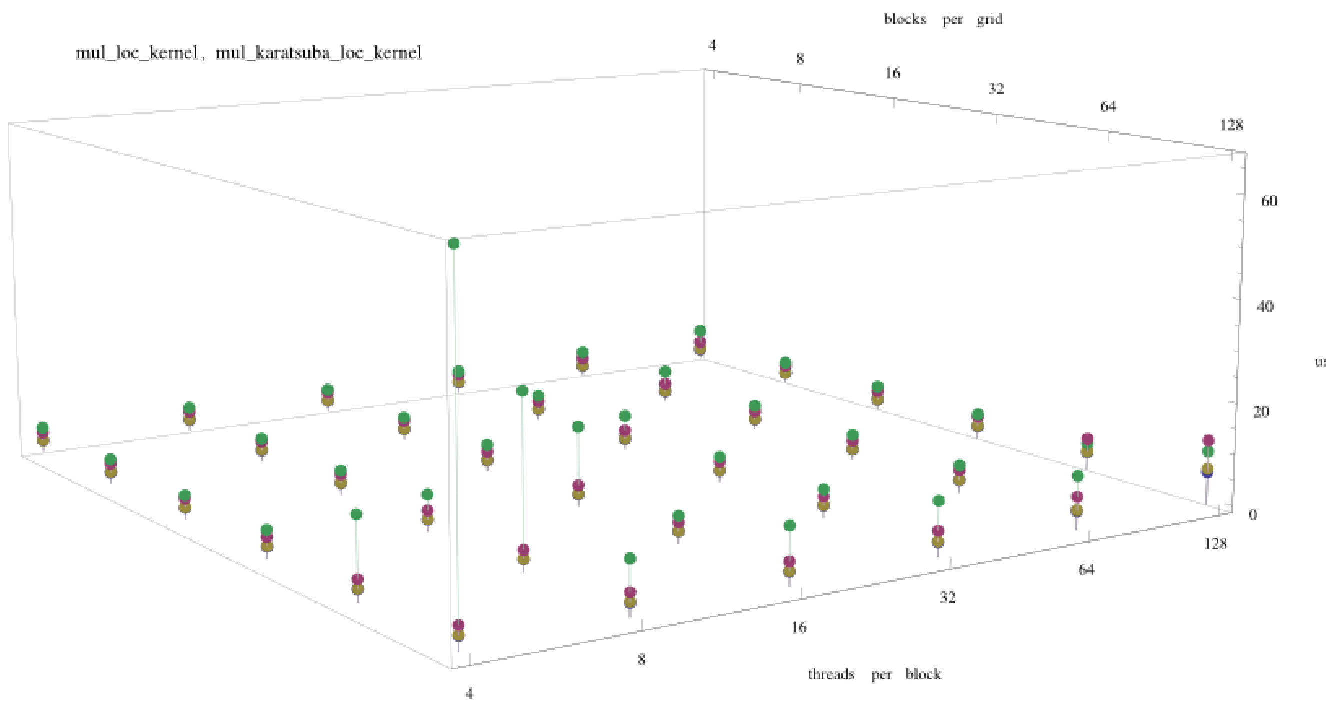
- Involve 10 consecutive executions of an operation.
- Types of benchmarks
  - Assembly vs. C (addition kernel only)
  - Local vs. global memory access
  - Importance of operand layout in memory
  - Classical vs. Karatsuba multiplication
  - Kepler vs. Fermi execution time
  - Warp occupation

Warp Occupation Results

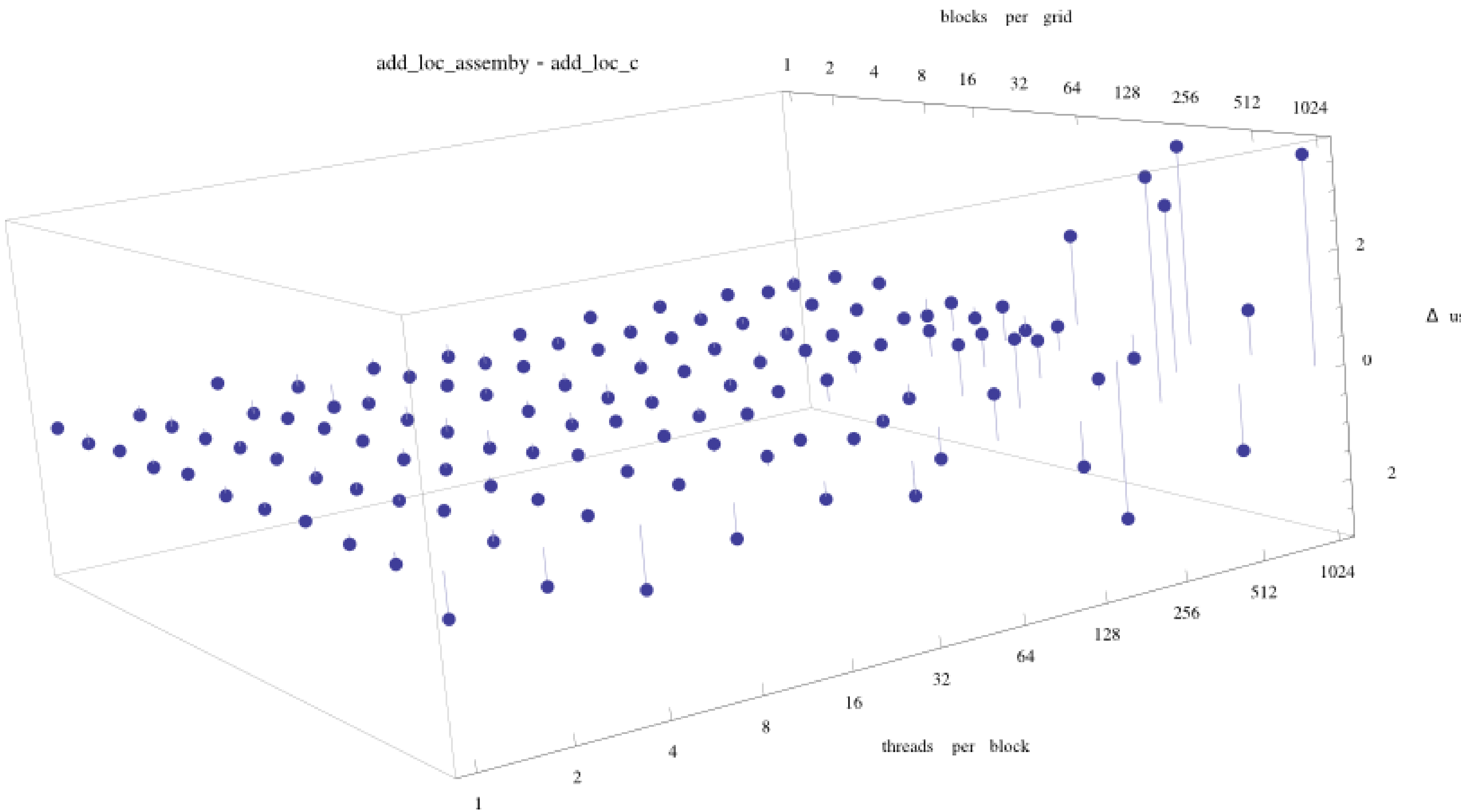
- Better to have less blocks, where more threads are used.
- Must keep warp occupation full, at least 32 threads.
- If not, then resources are wasted.

Karatsuba vs. Classical Multiplication and Warp Occupancy

- mul_loc 131-bit
- mul_karatsuba_loc 131-bit
- mul_loc 239-bit
- mul_karatsuba_loc 239-bit



Assembly vs. C



Operator verification

- Like benchmark infrastructure
- Operation results recovered from server
- Checked for errors with GMP on host

Conclusion

- Project was tedious to start
- Hard time debugging
- First hands-on experience
- Great fun learning

… it was only the beginning

- Montgomery multiplication
- Modular exponentiation
- Modular inversion

Use arithmetic to implement some algorithms (RSA, Pollard's Rho, …)

Questions?

Thanks