Workshop: School of Data Science and Computation Thinking, Stellenbosch University

Introduction to Parallel Computation with GPUs: Programming with JAX, NUMBA, THURST, and CUDA

Martin BUCHER
(bucher@sun.ac.za)
LECTURE 4 (27 Feb 2025)

Course dates (4-5pm SAST):

Tu 18 Feb	Th 20 Feb
Tu 25 Feb	Th 27 Feb
Tu 4 March	Th 6 March
Tu 11 March	Th 13 March

Zoom link:

https://u-paris.zoom.us/j/82740807191?pwd=8yAGlP3R6VKbsjNCIeRjJU3u0AQtU6.1

Course webpage:

http://www.sun.ac.za/english/data-science-and-computational-thinking/parallelcomputation
https://github.com/martinabucher/cudaCourse

Lecture 3 Outline

1. Computer Accounts – Status
2. Assignment
3. Probing Device Properties

```
clockRate                   = 2550000 (2.55e+06)
l2CacheSize                 = 75497472 (7.54975e+07)
major                       = 8 (8)
minor                       = 9 (9)
maxBlocksPerMultiProcessor  = 24 (24)
maxGridSize[0]              = 2147483647 (2.14748e+09)
maxGridSize[1]              = 65535 (65535)
maxGridSize[2]              = 65535 (65535)
maxThreadsDim[0]            = 1024 (1024)
maxThreadsDim[1]            = 1024 (1024)
maxThreadsDim[2]            = 64 (64)
maxThreadsPerBlock          = 1024 (1024)
maxThreadsPerMultiProcessor = 1536 (1536)
memPitch                    = 2147483647 (2.14748e+09)
memoryBusWidth              = 384 (384)
memoryClockRate             = 10501000 (1.0501e+07)
multiProcessorCount         = 128 (128)
name[256]                   = NVIDIA GeForce RTX 4090
persistingL2CacheMaxSize    = 51904512 (5.19045e+07)
regsPerBlock                = 65536 (65536)
regsPerMultiprocessor       = 65536 (65536)
sharedMemPerBlock           = 49152 (4.915200e+04)
sharedMemPerBlockOptin      = 101376 (1.013760e+05)
sharedMemPerMultiprocessor  = 102400 (1.024000e+05)
totalConstMem               = 65536 (6.553600e+04)
totalGlobalMem              = 25393692672 (2.539369e+10)
warpSize                    = 32
```

Limitations of sharing a common instruction stream

In the following all processors execute both branches when at least one but not all cores in the warp satisfy the condition.

```
if (condition) {
....
code
...
} else {
....
code
...
}
```

or here all the cores must wait until all have broken out of the endless loop

```
while(1){
....
code
...
if (condition) break;
....
code
...
}
```

Analysis of Algorithms

- If we consult a standard alorithms book (e.g., Cormen, Leiserson, Rivest and Stein Introduction to Algorithms, 4th Edition, MIT Press 1312 pages), we see the performance of algorithms expressed using $O(\cdot)$.
- On a CPU, for example a vector dot computation is $O(N)$, the multiplication of a vector by a matrix $O(N^2)$, matrix multiplication is $O(N^3)$.
- Mathematically, for integer functions

  $$O\big(f(n)\big) \sim O\big(g(n)\big)$$

  if

  $$\frac{f(n)}{g(n)}$$

  have upper and lower limits that are finite and greater than zero, respectively.
- This coarse-grained analysis of algorithms is used rather than a more precise analysis because it is independent of language and hardware details.

Analysis of Algorithms

- The $O(1)$ factors are important, to be sure, and worth working toward minimizing, but for example an $O(N)$ algorithm will always be superior to an $O(N\log(N))$ algorithm.
- Most analysis of algoritms assumes a single CPU machine (idealized to have an unlimited amount of memory, with uniform memory access).

Analysis of Parallel Algorithms

- Reference: Joseph JaJa, Introduction to Parallel Algorithms, 1992 Addison-Wesley
- Basic idealized hardware setep = PRAM (Parallel Random Access Machine)
  - Synchronous
  - Infinite number of available processors
  - Uniform time for access to memory

What is the class (difficulty) of the following problem on a parallel processor?

- Problem

$$\sum_{n=0}^{2^D} f(n)$$

  where $N, D \to \infty$.
- How would we implement it on an idealized PRAM? On CUDA? What kind of special parallel support is needed?

Reduction (I)

"Reduction" = Parallel Computation of Sums

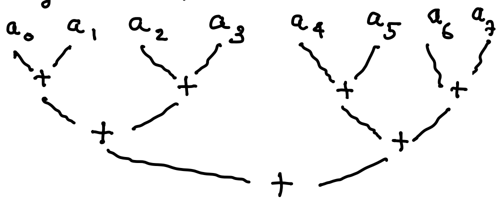
How long does it take to compute the following?

$$a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7$$

Associativity allows to regroup

$$\{(a_0 + a_1) + (a_2 + a_3)\} + \{(a_4 + a_5) + (a_6 + a_7)\}$$

Diagramtically:



$$\{\text{Time}\} \sim \lceil \log_2 N \rceil$$

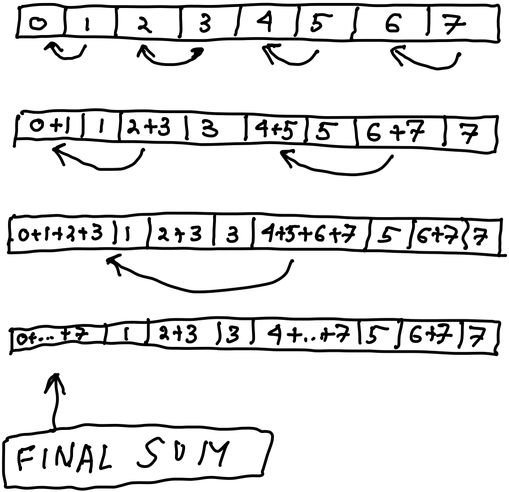
Reduction (II)

Reduction applies to any binary associative operator. E.g.

$+$ $\times$

AND OR

Reduction (II)



Reduction Kernel (summation)

```
#include <stdio.h>
__global__ void testReduceSum(float* s, const int n, float *result);

__device__ void reduce_sum(float *s, const int n, float *sum){
  if (n == 1){   // eliminate trivial case
     *sum = s[0];
  }
  int index = threadIdx.x + blockDim.x * blockIdx.x; int step=1;
  while(1){
    if (2*step >= n ) break;
    step*=2;
  }
  if (index == 0 )
     printf("step %d\n",step);
  int n_max=n;
  while(1){
    if ( index + step < n_max )
      s[index]+=s[index+step];
    __syncthreads();
    n_max=step;
    if ( step==1 ) break;
    step/=2;
  }
  *sum=s[0];
}
```

```
int main(void){
  const int thread_dim=10;
  const int block_dim =10;
  int n=100;
  size_t size=n*sizeof(float);
  float* s_d;
  float *result_d;
  float result_h;
  cudaMalloc((void **) &s_d, size);
  cudaMalloc((void **) &result_d,sizeof(float));
  float s_h[size];
  float local_sum=0.;
  for(int j=0;j<n;j++){
    s_h[j]=(float) (j+1);
    local_sum+=s_h[j];}
  printf("Expected sum is equal to %f \n", local_sum);
  cudaMemcpy(s_d, s_h, size, cudaMemcpyHostToDevice);
  testReduceSum<<<block_dim,thread_dim>>>(s_d,n,result_d);
  cudaMemcpy(&result_h, result_d, sizeof(float), cudaMemcpyDeviceToHost);
  printf("CUDA sum is equal to %f \n", result_h);
}
```

```
bash-4.2$ ./my_reduce
Expected sum is equal to 5050.000000
step 64
CUDA sum is equal to 5050.000000
bash-4.2$
```

Challenges of Sharing Data

- Suppose that we do a lot of work by different cores and the end result is to be sum of all the partial results. Suppose moreover that the most of the work is in the computation of the partial results and not in the computation of the sum. This differs from our discussion of reduction where the object was to attain maximum speed for computing a sum.
- We want to carry out an operation of the sort

  ```
  sum=sum+new_contribution;
  ```

  where the variable sum is shared between many processes.
- One of the problems is that the above operation when translated into machine instructions involves more than one step, even though in C/C++ it appears as a single statement.
  1. Retrieve sum (from main memory) to a register.
  2. Add new_contribution to the value of sum in the register.
  3. Write the new value back to memory.
- This needs to occur in such a way that only one processor at a time accesses the variable sum.
- One solution is to use a lock aka mutex=(mutual exclusion).

```cpp
#include <chrono>
#include <iostream>
#include <map>
#include <mutex>
#include <string>
#include <thread>

std::map<std::string, std::string> g_pages;
std::mutex g_pages_mutex;
void save_page(const std::string& url){
    // simulate a long page fetch
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::string result = "fake content";

    std::lock_guard<std::mutex> guard(g_pages_mutex);
    g_pages[url] = result;}

int main(){
    std::thread t1(save_page, "http://foo");
    std::thread t2(save_page, "http://bar");
    t1.join();
    t2.join();
    // safe to access g_pages without lock now, as the threads are joined
    for (const auto& [url, page] : g_pages)
        std::cout << url << " => " << page << '\n';}
```

Output:

```
http://bar => fake content
```

C++ interlude

- Here the C++ main program creates two threads running on different cores.
- In C++ the mantra is "RAII" or "Resource Acquisition Is Initialization". The idea is that the root of all evil is uninitialized variables or forgetting to release ressources once they are not needed.
- Thus when a variable is declared a "constructor" is called that initializes the variable, and when the variable falls out of scope, a destructor" is called to carry out any needed cleanup action.
- We already recounted how the founders of Java considered pointers the "root of all evil", replacing pointers with references and garbage collection. RAII is in part Bjarne's Stroustrup's reply to this criticism.
- It is worth looking at

  `https://en.cppreference.com/w/cpp/language/raii`

  where this ideology has been made part of the C++ standard library documentation.

The bad and the good according to the C++ gurus

```
std::mutex m;
void bad(){  // THIS IS HOW ONE WOULD PROGRAM THIS IN OLD-STYLE C
    m.lock();  // acquire the mutex
    f();       // if f() throws an exception, the mutex is never released
    if (!everything_ok())
        return; // early return, the mutex is never released
    m.unlock(); // if bad() reaches this statement, the mutex is released
}

void good(){ // THIS IS THE C++ WAY OF DOING THE SAME
    std::lock_guard<std::mutex> lk(m);
        // RAII class: mutex acquisition is initialization
    f();       // if f() throws an exception, the mutex is released
    if (!everything_ok())
        return;// early return, the mutex is released
}        // if good() returns normally, the mutex is released
```

One cannot forget to include releasing the lock when writing the program, because this is done implicity at the end of subroutine.

https://en.cppreference.com/w/cpp/language/raii

Not everyone loves C++

Source: Reddit

What can go wrong (I): Deadlocks (part 1)

- In general, function calls to acquire locks are **blocking**, which means that they do not return until the lock is acquired. In other words, if someone else has the lock, the routine waits. Normally, after a while the thread would be swapped out of the CPU or core, assuming that there are more threads than cores.
- Another option is for the function call attempting to acquire the lock to return with a failure status.
- Worst Case Scenario (but not so uncommon if one is not careful)
  Consider the two processes running on different threads:

```
process_1(){
   get_lock_A();
   get_lock_B();
   do_work_1();
   release_lock_B();
   release_lock_A();
}

process_2(){
   get_lock_B();
   get_lock_A();
   do_work_1();
   release_lock_A();
   release_lock_B();
}
```

What can go wrong (I): Deadlocks (part 2)

- The two (asynchronous) streams may interleave arbitrarily. For example, the following chronological sequence is possible:

  ```
  process_1 acquires lock A.
  process_2 acquires lock B.
  ```

- Now `process_1` is waiting for lock B to be released and `process_2` is waiting for lock A to be released. Both get_lock calls are blocked, each waiting for something to happen that will never happen.
- The correctness of a parallel program relies on it being able to run correctly and finish for all possible interleaving of the execution chronology.
- What are some solutions to this problem?

Some more considerations:

- "Interleaving" not mean what you think it does.
- The simplest interpretation would be to order sequences of code (say for two processes A and B: as A1, A2,....,AM and B1, B2, ..., BN and enumerate of sequences of length M+N where all the As appear exactly once respecting the A order and all the Bs appear exactly once respecting the B order.
- But these operations may not be atomic, so finer-grained overlap or simultaneity may occur.
- An example is

  ```
  sum=sum+term;
  ```

Livelock (I)

```
process_1(){
  bool isHungry=true;
  while(isHungry){
    if (get_lock_A(){
       if (get_lockB() ){
         do_work_1();
         is_hungry=false;
         release_lock_B()
       }
    } else {
      release_lock_A();
    }
  }
}

process_2 is the same except that
B and A are interchanged and the work is
different.
```

Livelock (II)

An infinite loop is created, and nothing gets done. Each tries to defer to the other, but the end result is not successful.

```
process 1 acquires lock A
process 2 acquires lock B
process 1 fails to get lock B
process 2 fails to get lock A
process 2 releases lock B
process 1 releases lock A

process 1 acquires lock A
process 2 acquires lock B
process 1 fails to get lock B
process 2 fails to get lock A
process 2 releases lock B
process 1 releases lock A

ad infinitum
```

Race Conditions

- The final result depends on how the asynchronous execution is interleaved.
- The outcome of the race is not necessary predictable. It may depend on the hardware or on the data.
- These kinds of errors are hard to find and may pop up after the software appears to have been succesfully tested.

Atomic Operations (I): Atomic CAS (Compare and Swap)

```
int compare_and_swap(int *destination_address, int expected_value,
int desired_value);

The following operation is carried out atomically.

If the destination address value is thz expected value, it is replaced
by the desired value, which is also value returned. Otherwise, the
present value is returned.

Usage exam:

int sum=0;
int expected_value;
do { expected_value=sum;
     desired_value=expected_value+partial_sum;
} while(desired_value!=compare_and_swap(&sum, expected_value, desired_value);

When there is contention at least one thread succeeds.
```