> By Jiahui Tang

Spec: (Same as 3.1)

- GPU: AWS GPU g3.4xlarge, Tesla M60 (More details attached)
- OS: Ubuntu 18.04
- Kernel: Linux ip-172-31-36-197 5.4.0-1039-aws #41~18.04.1-Ubuntu SMP Fri Feb 26 11:20:14 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
- EBS: Resize EBS Volume to 128GB
- Compiler: gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0 (no flags)

```
ubuntu@ip-172-31-36-197:~$ uname -a
Linux ip-172-31-36-197 5.4.0-1039-aws #41~18.04.1-Ubuntu SMP Fri Feb 26 11:20:14 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```

```
ubuntu@ip-172-31-36-197:~$ lspci | grep -i nvidia
00:1e.0 VGA compatible controller: NVIDIA Corporation GM204GL [Tesla M60] (rev a1)
ubuntu@ip-172-31-36-197:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev             60G     0   60G   0% /dev
tmpfs            12G   844K  12G   1% /run
/dev/xvda1      7.7G   1.2G  6.6G  15% /
tmpfs            60G     0   60G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs            60G     0   60G   0% /sys/fs/cgroup
/dev/loop0       32M    32M     0 100% /snap/snapd/11036
/dev/loop1       56M    56M     0 100% /snap/core18/1988
/dev/loop2       34M    34M     0 100% /snap/amazon-ssm-agent/3552
tmpfs            12G     0   12G   0% /run/user/1000
ubuntu@ip-172-31-36-197:~$
```

```
ubuntu@ip-172-31-36-197:~$ pgaccelinfo

CUDA Driver Version:           11020
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  460.32.03  Sun Dec 27 19:00:34 UTC 2020

Device Number:                 0
Device Name:                   Tesla M60
Device Revision Number:        5.2
Global Memory Size:            7988903936
Number of Multiprocessors:     16
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    873 MHz
Execution Timeout:             No
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   Yes
Memory Clock Rate:             2505 MHz
Memory Bus Width:              256 bits
L2 Cache Size:                 2097152 bytes
Max Threads Per SMP:           2048
Async Engines:                 2
Unified Addressing:            Yes
Managed Memory:                Yes
Concurrent Managed Memory:     No
```

```
ubuntu@ip-172-31-36-197:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.5.0-3ubuntu1~18.04' --with-bugurl=fi
only --program-suffix=-7 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id
e=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-
ib --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --wi
cking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
ubuntu@ip-172-31-36-197:~$
```

```
ubuntu@ip-172-31-36-197:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
stepping        : 1
microcode       : 0xb000038
cpu MHz         : 1794.171
cache size      : 46080 KB
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge m
cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l
bogomips        : 4600.14
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

3.2. Acceleration of Matrix Multiplication (15 points)

Develop an OpenACC version of `seq_mm.c` and compare it with the optimized sequential execution of section 2.2.

Submission

- `P32.pdf`: Discuss the implementation, its performance and the default block grid created by the system
- `P32.c`: Source code

```
ubuntu@ip-172-31-36-197:~$ ./p32
****************************************************
||c||_F = 2630354827429312.00
Elapsed Time: 0.503553 s.

Accelerator Kernel Timing data
/home/ubuntu/P32.c
  main  NVIDIA  devicenum=0
    time(us): 217,043
    37: compute region reached 1 time
        40: kernel launched 1 time
            grid: [17579]  block: [128]
             device time(us): total=210,533 max=210,533 min=210,533 avg=210,533
            elapsed time(us): total=210,588 max=210,588 min=210,588 avg=210,588
    37: data region reached 2 times
        37: data copyin transfers: 6
             device time(us): total=4,882 max=1,507 min=120 avg=813
        45: data copyout transfers: 2
             device time(us): total=1,628 max=1,502 min=126 avg=814
ubuntu@ip-172-31-36-197:~$
```

Since the spec of this question changed, we rerun optimized flags for original version, and also rerun the blocking/unrolling optimized sequential execution from section 2.2

	-O0	-O1	-O2	-O3	-Ofast (*in gcc, not pgcc -acc*)	No optimization, reference	OpenAcc	blocking/unrolling without flags
seq_mm.c	15.1101	14.5634	1.28416	1.28362	3.64143	6.93109	0.503553	2.45214

Discussion:

Implementation

```
get_time(&tstart);

#pragma acc data copy(c[0:N][0:N]) copyin(a[0:N][0:N]) copyin(b[0:N][0:N])
#pragma acc kernels loop independent
//#pragma acc parallel loop independent
for (i=0; i<N; i++)
  {
  for(j=0; j<N; j++){
    for (k=0; k<N; k++) {
      c[i][j] += a[i][k] * b[k][j];
      }
    }
  }

   get_time(&tend);
```

`#pragma acc data copy(c[0:N][0:N]) copyin(a[0:N][0:N]) copyin(b[0:N][0:N])`

- matrix c: It is defined before the for loop, and will be changed inside the inner for loops, and will also be an output after executing for loops, so we use copy clause for both copy in and

export.

- matrix a,b: the a,b matrix information are used when running inner for loops, so we need copyin to send its data into the parallel processing. But there's no output for these two matrix needed, as their value are not changed. So we use only `copyin` instead of `copy`.
- [0:N][0:N]: since we want to make sure that length of space is appropriately allocated for the elements in matrix, we explicitly specify the matrix index and length.

`#pragma acc kernels loop independent` (similar to 3.1)

I tried to distinguish the difference between two different clauses

- #pragma acc kernels loop
- #pragma acc parallel loop

They could all be compiled but the main difference is that kernels construct may be thought of as a hint to the compiler of where it should look for parallelism, while the parallel directive is an assertion to the compiler of where there is parallelism. Thus, I put it as `#pragma acc kernels loop`.

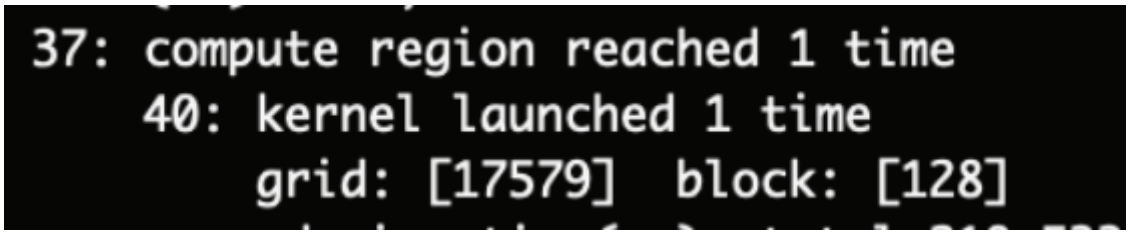
As for `independent`, by using this construct in conjunction with the loop construct, we can tell the compiler that loop iterations are data-independent and can be executed in parallel, overriding compiler dependency analysis.

Performance

Looking at the performance of `seq_mm.c`, the OpenAcc performance much **better** than all the optimized sequential execution version in section 2.2.

As OpenAcc version uses GPU programming, and explicitly expressed parallelism in the code, it did accelerated computing. Section 2.2 only did performance optimization with optimization flag and manual optimization of unrolling/blocking, GPU programming with OpenAcc could massively parallel with hundreds of cores and thousands of threads. And loop here is large, large independent loops could offset GPU/memory overhead. Overall, GPU-accelerated computing offloads compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run much faster. Thus it performs much better than all the rest in section 2.2

Default Block Grid



The default block here is `128`, grid size is `17579`.

Here by using the default block `128`, and grid being `17579`, the number of gangs is `17579/128=137.335`, and it keeps all warps busy. In the code, we can see matrix size `N =`

`1500` , indicating the matrix size (2D) is `1500 x 1500 = 225000` . Here the block size multiplies the grid is roughly the same as the problem size.

By research, we know that choosing the execution configuration parameters should be done in tandem; however, there are certain heuristics that apply to each parameter individually. When choosing the first execution configuration parameter -the number of blocks per grid, or grid size - the primary concern is keeping the entire GPU busy. Here the number of blocks in a grid should be larger than the number of multiprocessors so that all multiprocessors have at least one block to execute.

Reference: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

In [ ]: