Compilers and Libraries

Dr. Dimitris Dellis
ntell@grnet.gr

GRNET

Athens, 10-11 Feb 2020

Outline I

- Compilers and Switches
- Libraries
  - Various CPU libraries
  - various GPU libraries
- CPU vs GPU Libraries vs data sizes
- How to easy (or not) port your application to GPUs

Compilers and Compilation Flags I

- CPU Compilers and Flags
  - GNU (versions)
  - Intel (versions)
  - PGI (versions)
  - Native CUDA (NVIDIA and versions)

```
nvcc -O3 file.cu --gpu-architecture=compute_35 \
     --gpu-code=compute_35,sm_35 -o file.exe
```

Compilers and Compilation Flags I

An example : Matrix - Matrix Multiplication

- Some definitions

$$\begin{bmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1K} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2K} \\ .. & .. & \dots & .... \\ \alpha_{M1} & \alpha_{M2} & \dots & \alpha_{MK} \end{bmatrix} x \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \dots & \dots & \dots & \dots \\ x_{K1} & x_{K2} & \dots & x_{KN} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1N} \\ b_{21} & b_{22} & \dots & b_{2N} \\ \dots & \dots & \dots & \dots \\ b_{M1} & b_{M2} & \dots & b_{MN} \end{bmatrix}$$

Compilers and Compilation Flags I

An example : Matrix - Matrix Multiplication

- Requires : K multiplications + K additions for each element of B.
- Total $2N \cdot M \cdot K$ floating poing operations.
- For M=N=K=1000 : $2 \cdot 10^9$ operations.
- If a machine is able to perform the multiplication in 1 sec then its performance is 2 GFLOPS. In next sections, we'll use the GFlops rate for comparison.

Compilers and Compilation Flags I

An example : Matrix - Matrix Multiplication

- Examples in ForTran (Discussion about Fortran vs C/C++)

```
do k = 1,n1
   do i = 1,n3
     c(i,k) = 0.0
     do j = 1,n2
        c(i,k) = c(i,k) + a(i,j) * b(j,k)
     enddo
    enddo
enddo
```

Compilers and Compilation Flags I

An example : Matrix - Matrix Multiplication

- For those who complain about Fortran, equivalence between C and Fortran is :

```
for(i=A;i<B;i+=C) { }        do i=A,B,C enddo
if(x<y){ }                   if(x.lt.y)then endif
x[i][j]                      x(i,j)
/* or //                     any letter in 1st column
pow(x,y)                     x**y
```

Compilers and Compilation Flags I

An example : Matrix - Matrix Multiplication

- Switch to interactive code view : See the code.

Compilers and Compilation Flags I

- Common choices (if available) :
- GNU : Select the higher available version
- Intel : Select the higher available version
- PGI : Select the higher available version

Compilers and Compilation Flags I

- GNU
  - Try optimization flags
  - `-O3 -march=native -mtune=native`
  - (Switch to interactive)
- Intel
  - Try optimization flags
  - `-O3 -xHost`
  - (Switch to interactive)
- PGI
  - Try optimization flags
    - `-O4 -tp=haswell -Mvect=altcode,fuse,gather,simd,256,tile`
    - `-m64 -Mconcur=allcores,assoc -Mpropcond -Minfo=all`

Compilers and Compilation Flags II

- (Switch to interactive)
- Keep in mind Cross compilation, i.e. compile on a machine with optimizations for another machine.

Compilers and Compilation Flags I

- Try No optimization : i.e
- `gcc/gfortran file.c -o file.exe`
- `icc/ifort -O0 file.c -o file.exe` (intel has default -O2 without specifying optimization level
- `pgcc/pgfortran -O0 file.c -o file.exe`

Compilers and Compilation Flags I

- Let's see performance of this piece of code without any direct or indirect optimization :
- Performance highly depends on the data size i.e. arrays dimension.

Compiler	Flags	Time to perforrm 5 times the MM for N=8192 [*sec*]
GNU 8.2.0	-	66798.68
PGI 18.10	-	42063.00
Intel 18.0.2	-O0	2400.709

Compilers and Compilation Flags II

- It seems that Intel has greater performance, PGI follows, last is GNU.
- Let’s switch to compiler optimization flags, and use of OpenMP/Threads auto parallelization.

Compilers and Compilation Flags

- Max Optimization Flags

Compiler	Flags	Time to perform 5 times the MM for N=8192 [*sec*]
GNU 8.2.0	-O3 -mavx2 -mfma -march=haswell -mtune=haswell -fexpensive-optimizations	7805.12
PGI 18.10	-O4 -tp=haswell -Mvect=altcode,fuse,gather,simd,256,tile -m64 -Mconcur=allcores,assoc -Mpropcond -Minfo=all	191.33
Intel 18.0.2	-O3 -xCORE-AVX2 -unroll -unroll-aggressive	320.88

Compilers and Compilation Flags

- Up to now : From 1113.3 minutes on a single core using no optimization with GNU went to 5.35 min with Intel on a single core or 3.19 min with 20 cores and PGI.
- Speed up : 208.1 on a single core, or 350 using full 20C core node.
- It is enough ?
- NO.
- What can we do without any change to code (Indeed no algorithm change)

Compilers and Compilation Flags

- OpenACC : Try to use compiler and conditional directives to improve performance using GPUs.
- With PGI add `-ta=nvidia -acclibs -acc=autopar` in PGI flags toghether with some code directives
-
```
!$acc kernels
      do k = 1,n1
           ...
      enddo
!$acc end kernels
```
- Switch to interactive

Compilers and Compilation Flags

- With these PGI flags that automatically = no code changes except addion of acc directives, very similar to OpenMP, we get

Compilers and Compilation Flags

- OpenACC/PGI Performance

Compiler	Flags	Time to perform 5 times the MM for N=8192 [*sec*]
PGI 18.10	-O4 -tp=haswell -Mvect=altcode,fuse,gather,simd,256,tile -m64 -Mconcur=allcores,assoc -Mpropcond -Minfo=all -ta=nvidia -acclibs -acc=autopar	88.68

Compilers and Compilation Flags

- Without any algorithm/code change except addition of 2 directive lines we went from 1113.3 minutes to 1.8 minutes, a speed up of 753.
- It is enough ?
- NO

Use of optimized Libraries

- Libraries : Pieces of code that perform certain tasks and they are usuallu optimized of almost any underlying architecture.
- Requires to port once the code to use them.
- Then we can use almost any architecture via its optimized libraries.
- For Matrix-Matrix Multiplication (and many other Linear Algebra operations), the BLAS functions are good candidates.
- Let’s start with necessary code changes.

Use of optimized Libraries I

- We have to replace the ”in code triple loop” matrix-matrix multiplication with a function call :

```
call dgemm('N','N',n1,n2,n3,ONE,a,lda,b,ldb,ZERO,c,ldc)
```

- There are various CPU/GPU/Phi implementations of dgeem (and many other BLAS functions) optimized on certain architectures.
- We ’ll focus on CPU and GPU implemtations.
- Some BLAS variants :
  - Old (dated back to 1977, still improving, a reference) netlib blas (and lapack).
  - OpenBLAS : A best guess tuned BLAS (and lapack) library.

Use of optimized Libraries II

- ATLAS : Fully autotuned during compilation BLAS (and lapack library)
- MKL : Intel BLAS (and lapack among others) autotuned library.
- NVBLAS : NVIDIA Optimized BLAS library that runs on NVIDIA GPUs
- Other alternatives are magma (autoswitch betwenn CPU or GPU) and some other fresh implementations not ready available that use the underlying available versions of library.

Use of optimized Libraries

- Using various BLAS libraries the timing of Martix-Matrix multiplication (including single core code timing) is :

BLAS Impl.	Time to perform 5 times the MM for N=8192 [*sec*]
Code/GNU no optim.	66798.68
Code/Intel high opt. level	320.88
Netlib (Single Core)	1939.44
OpenBLAS(Single Core)	279.11
OpenBLAS(OMP, 20 cores)	14.89
MKL(single core)	140.55
MKL(SMP, 20 cores)	16.72
ATLAS(SMP, 20 cores)	14.43
NVBLAS(1K40m, single core)	4.91

Use of optimized Libraries I

- Going from Code to Library, using automatic CUDA BLAS implementation, we obtain a speed up of $\sim$ 13,900 i.e. $\sim$ 13,900 times faster than the usual in code implementation.
- If you use CPU optimized libraries, you get speed up close to theoretical speedup of GPU vs node performance, depending on available CPU and GPU in the range 2-6. Compare performance of ATLAS/OpenBLAS/MKL SMP vs NVBLAS.
- This applies to large arrays dimension. What happens with small arrays ?

Use of optimized Libraries II

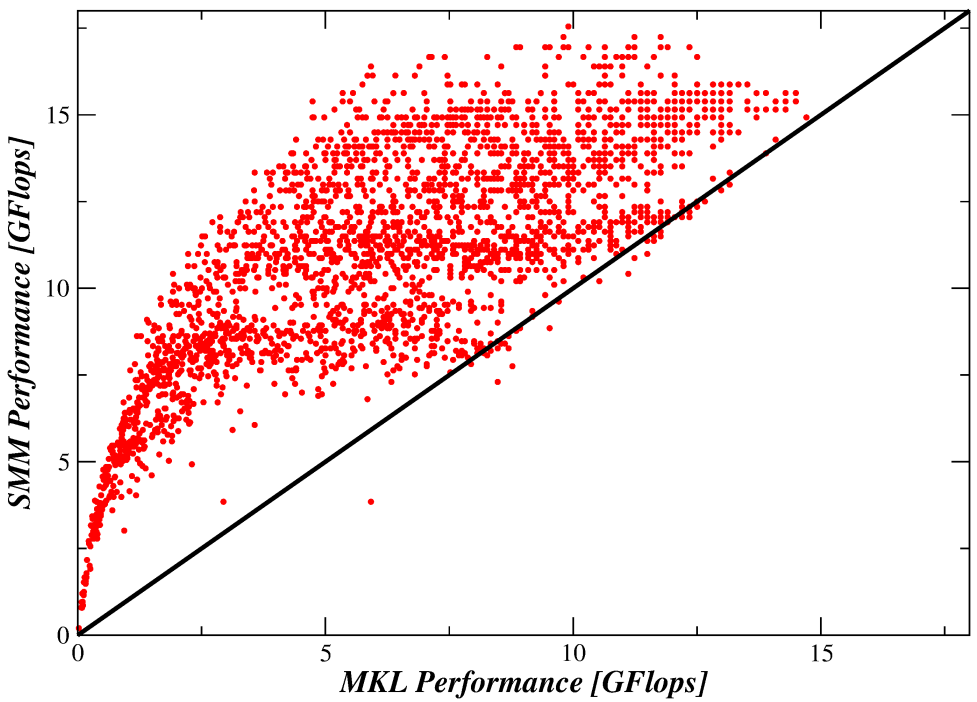
- A very common matrix-matrix multiplication with sizes 3x3 or other exotic sizes like 6x7, 11x7 etc. is not good candidate for accelerators, at least in the usual way, look for SMM.
  - Again we have to alter the code : Instead of

    ```
    call dgemm('N','N',NMAX,NMAX,NMAX,ONE,a,lda,b,ldb,ONE,c,ldc)
    ```

  - We have to modify it to look (for double precision) like:

    ```
    call smm_dnn(M,N,K,A,B,C)
    ```

Use of optimized Libraries



Questions ?