ISC17 SCC CODING CHALLENGE REPORT

SUBMITTED BY:
NANYANG TECHNOLOGICAL UNIVERSITY

CODE CONTRIBUTOR(S):
LIU SIYUAN

Contents

1 Optimizations

This section describes various optimization techniques team NTU employed to accelerate miniDFT.

1.1 Compilation optimization

We first optimized miniDFT by linking optimized libraries and using optimized compilers for our platform. In particular, we linked miniDFT against Intel Math Kernel Library (MKL) and used Intel compiler and Intel MPI to compile miniDFT. Also, with extensive testing, we discovered that MPI provides better parallelism than OpenMP and it is also more stable (OpenMP version gives wrong result sometimes). Therefore, we disabled OpenMP when compiling miniDFT (linking to `libmkl_sequential.a` and compile without `__OPENMP` and `-qopenmp`).

We also optimized compilation by tuning pre-processor macros. We discovered that there is a macro named `__NONBLOCKING_FFT` which controls what MPI routines are used for the `fft_scatter` function. The non-blocking version with asynchronous send and receive appears to perform much better on InfiniBand interconnect than the all to all version.

Below are the options we used in the Makefile for compilation optimization.

```
DFLAGS = -D__INTEL -D__FFTW -D_SCALAPACK -D__MPI -D__NONBLOCKING_FFT

MATH_LIBS = ${MKLROOT}/lib/intel64/libmkl_scalapack_lp64.a \
            -Wl,--start-group \
              ${MKLROOT}/lib/intel64/libmkl_intel_lp64.a \
              ${MKLROOT}/lib/intel64/libmkl_sequential.a \
              ${MKLROOT}/lib/intel64/libmkl_core.a \
              ${MKLROOT}/lib/intel64/libmkl_blacs_intelmpi_lp64.a \
            -Wl,--end-group -lpthread -lm -ldl

CC = mpiicc
CFLAGS = -O3 -I./ -I${MKLROOT}/include

FC = mpiifort
FFLAGS = -O3 -fpp -I./ -I${MKLROOT}/include

LD = mpiifort
LDFLAGS =
```

1.2 Code optimization

During our preparation, we tried to optimize the code by changing the algorithms and porting part of the code to CUDA. Below are the detailed descriptions to our code optimization.

1.2.1 Optimize algorithm in `add_vuspsi_k`

In the original code, `add_vuspsi_k` calls `calbec` to perform `ZGEMV` matrix vector multiplication calculations and do a sum reduce over the resulting vector (as shown below).

```fortran
DO ibnd = 1, m
    ! JRD: Compute becp for just this ibnd here
    !
    CALL calbec ( n, vkb, psi, becp, ibnd )
    !write(*,*) 'Computing becp', ibnd
    !
    ! other code ...
    !
END DO
```

```fortran
!-----------------------------------------------------------------------
SUBROUTINE calbec_k ( npw, beta, psi, betapsi, ibnd )
!-----------------------------------------------------------------------
    !
    ! other code ...
    !
    CALL ZGEMV( 'C', npw, nkb, (1.0_DP,0.0_DP), beta, npwx, psi(:,ibnd), 1, &
                (0.0_DP, 0.0_DP), betapsi, 1 )
    CALL mp_sum( betapsi( : ), intra_bgrp_comm )
    !
    CALL stop_clock( 'calbec' )
    !
    RETURN
    !
END SUBROUTINE calbec_k
```

We noticed that the loop in `add_vuspsi_k` is actually calling `calbec` to perform ZGEMV operations over the matrix `psi`. Each iteration in the loop corresponds to one column in the matrix. This is equivalent to performing a single `ZGEMM` matrix matrix multiplication on `vkb` and `psi`. As `ZGEMV` is memory-bound and `ZGEMM` is compute-bound, we obviously prefer to use `ZGEMM`.

In our optimized version of `add_vuspsi_k`, it performs a single `ZGEMM` operation instead of calling `calbec` in the loop. And then it performs the sum reduce over the resulting matrix (instead of vector). This reduced 8488 calls to `ZGEMV` to 5 calls to `ZGEMM` for the input `pe-23.LOCAL.in`.

```fortran
USE mp_global, ONLY : intra_bgrp_comm
USE mp,        ONLY : mp_sum
USE wvfct,     ONLY : npwx
!
! other code ...
!
COMPLEX(DP), ALLOCATABLE :: betapsi(:,:)
!
! other code ...
!
ALLOCATE( betapsi(SIZE(vkb,2), m) )
betapsi(:,:) = (0.0_DP,0.0_DP)
CALL ZGEMM('C', 'N', SIZE(vkb,2), m, n, (1.0_DP,0.0_DP), vkb, npwx, &
           psi, lda, (0.0_DP,0.0_DP), betapsi, SIZE(vkb,2))
CALL mp_sum(betapsi(:,:), intra_bgrp_comm)

DO ibnd = 1, m
    ! nested loop ...
    ps(ikb,ibnd) = ps(ikb,ibnd) + &
        deeq(ih,jh,na,current_spin) * betapsi(jkb,ibnd)
END DO
```

1.2.2 Accelerate BLAS

As mentioned in Section 1.2.1, `?GEMM`s are compute-bound routines. Therefore, they are perfect targets for offloading computation to GPUs. In our case, we used the Fortran thunking interface to cuBLAS provided by Nvidia to offload `ZGEMM` to GPUs. We decided to offload calls to `ZGEMM` in `cegterg.f90` and `add_vuspsi.f90` (after our optimization in Section 1.2.1). Some macros help us easily switch between MKL BLAS and cuBLAS.

```
#if defined(__CUDA) && defined(__CUBLAS)
#define ZGEMM cublas_ZGEMM
#endif
```

As we disabled OpenMP, we run one MPI rank per CPU core. This means that we would launch a huge number of processes on a single GPU. As they all have different CUDA context, the calls to GPU get serialized between those processes. In order to overcome this problem, we used Nvidia Multi-Process Service (MPS). Nvidia MPS allows multiple processes to share a CUDA context on the same GPU. The efficiency of `ZGEMM` are greatly improved on the GPU after using Nvidia MPS.

1.2.3 Accelerate (Sca)LAPACK

In the original code, ScaLAPACK is used in `cdiaghg.f90` to perform the diagonalization. We decided to port the diagonalization to GPU. As we are not aware of any mature ScaLAPACK implementation on the GPU, we used MAGMA, a serial hybrid (CPU + GPU) LAPACK library.

Because we used a serial LAPACK library, we have to create a serial version of `pcdiaghg` (named `cdiaghg_gpu`). We borrowed some code from the `qe-gpu-plugin` GitHub repository and fixed an error caused by the magic number inside. After the bug fix, the diagonalization could be successfully performed on the GPU with MAGMA. For each band group, only the root rank performs the diagonalization (hence making it serial). We used the single GPU interface (`magmaf_zhegvd`) instead of the multi-GPU interface because we realized that the single GPU version performs better for the matrix size we have to deal with.

```
IF ( me_bgrp == root_bgrp ) THEN
    CALL magmaf_init()
    !
    ! other code ...
    CALL magmaf_zhegvd(1, 'V', 'U', n, v, ldh, s, ldh, e, &
                       work, lwork, rwork, lrwork, iwork, liwork, info)
    !
    ! other code ...
    !
    CALL  magmaf_zhegvx( 1, 'V', 'I', 'U', n, h, ldh, s, ldh, &
                         0.D0, 0.D0, 1, m, abstol, mm, e, v, ldh, &
                         work, lwork, rwork, iwork, ifail, info )
    !
    ! other code ...
    !
    CALL magmaf_finalize()
END IF
!
! ... broadcast eigenvectors and eigenvalues to all other processors
!
CALL mp_bcast( e, root_bgrp, intra_bgrp_comm )
CALL mp_bcast( v, root_bgrp, intra_bgrp_comm )
```

1.2.4 Accelerate FFT

We tried to also port FFT to GPU with the cuFFT library. To make the porting process easy, we decided to use the PGI compiler (CUDA Fortran). We successfully ported all FFT operations to GPU, including the task group version, with CUDA-aware MPI. We also enabled GPUDirect RDMA on our cluster nodes to accelerate the communication as `fft_scatter` takes quite some time. In the end, however, we disabled our optimization on FFT because it appears to be slower than the CPU (88 CPU cores v.s. 8 Tesla P100s). Maybe it's because we have so many CPU cores and cuFFT cannot provide too much speed up compared to MKL FFT.

Interested readers could refer to `vloc_psi_gpu.f90` and other FFT related files with the suffix `_gpu.f90`.

1.2.5 Pinned memory for CUDA

Pinned memory is an optimization technique for accelerating data transfers between CPU and GPU. It allocates page-locked host memory to speed up the transfer. In our code, if PGI compiler is used, the large arrays in `pcegterg` are allocated as pinned memory. This optimization is not available if Intel compiler is used. In practice, we cannot notice a difference as the array are quite large and the benefit of pinned memory becomes negligible.

1.3 Runtime optimization

1.3.1 Command line options

We briefly studied the background of miniDFT to understand how the various command line options would impact the performance. With what we learned and our testing, it seemed that none of the command line options need to be adjusted. Optimizations like band group parallelism and task group parallelism seem to be beneficial only when running miniDFT at a very large scale (e.g. thousands of cores). If we enable those options manually, the performance will actually drop on our platform (small scale).

1.3.2 Process binding

As we ported part of the code to GPU, miniDFT now runs on a heterogeneous platform. Therefore, it's necessary to bind MPI processes to the CPU socket which is directly connected to the particular GPU the process controls in order to improve performance. We used `numactl` and bind processes according to their rank when launching the job (we have 4 GPUs in a single node).

```
#!/bin/bash

rank=$((${PMI_RANK} % 4))
case ${rank} in
[0]) numactl --cpunodebind=0 ./mini_dft -in pe-23.LOCAL.in;;
[1]) numactl --cpunodebind=0 ./mini_dft -in pe-23.LOCAL.in;;
[2]) numactl --cpunodebind=1 ./mini_dft -in pe-23.LOCAL.in;;
[3]) numactl --cpunodebind=1 ./mini_dft -in pe-23.LOCAL.in;
esac
```

2 Compilation and run

This section is meant to guide the reader to successfully compile and run our optimized version of miniDFT. First, please make sure the following pre-requisites are met. For compiling MAGMA, please refer to their documentation[1]. It's well documented and not hard to compile. For Fortran thunking interface and Nvidia MPS[2], please refer to Section 2.1 and 2.2.

- MAGMA (compiled with Intel compiler and MKL, without OpenMP)
- Intel compilers and Intel MPI
- GCC
- Intel MKL
- CUDA (with Fortran thunking cuBLAS interface)
- Nvidia MPS server (running on the node)
- numactl

To compile the code, use the following Makefile (available as a gist[3]) and type `make`. Make sure you have `MKLROOT` and `MAGMAROOT` as environment variables. Also, make sure you have `fortran_thunking.o`.

```
# macros
DFLAGS = -D__INTEL -D__FFTW -D_SCALAPACK -D__MPI -D__CUDA -D__MAGMA -D__ZHEGVD -
    D__CUBLAS -D__NONBLOCKING_FFT

# libraries
MATH_LIBS = ${MKLROOT}/lib/intel64/libmkl_scalapack_lp64.a \
            -Wl,--start-group \
              ${MKLROOT}/lib/intel64/libmkl_intel_lp64.a \
              ${MKLROOT}/lib/intel64/libmkl_sequential.a \
              ${MKLROOT}/lib/intel64/libmkl_core.a \
              ${MKLROOT}/lib/intel64/libmkl_blacs_intelmpi_lp64.a \
            -Wl,--end-group -lpthread -lm -ldl

```

[1] http://icl.cs.utk.edu/projectsfiles/magma/doxygen/installing.html
[2] https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf
[3] https://gist.github.com/koallen/da937a36b2f292e86a6e872e78354789

```
LIBS = ${MAGMAROOT}/lib/libmagma.a -lcuda -lcudart -lcublas -lcusparse $(
    MATH_LIBS) -lstdc++

CC = mpiicc
CFLAGS = -O3 -I./ -I${MKLROOT}/include

FC = mpiifort
FFLAGS = -O3 -fpp -I./ -I${MKLROOT}/include

LD = mpiifort
LDFLAGS =

CFLAGS += $(DFLAGS)

FFLAGS += $(DFLAGS)

OBJECTS += \
        gpu_aux.o \
        kind.o \
        constants.o \
        radial_grids.o \
        atom.o \
        io_global.o \
        parallel_include.o \
        mp.o \
        mp_global.o \
        cuda_global.o \
        parser.o \
        parameters.o \
        input_parameters.o \
        io_files.o \
        control_flags.o \
        cell_base.o \
        check_stop.o \
        clocks.o \
        fft_types.o \
        fft_base.o \
        fft_base_gpu.o \
        random_numbers.o \
        ions_base.o \
        descriptors.o \
        electrons_base.o \
        version.o \
        environment.o \
        error_handler.o \
        cufft.o \
        fft_scalar.o \
        fft_scalar_gpu.o \
        fft_custom.o \
        recvec.o \
        pwcom.o \
        stick_base.o \
        stick_set.o \
        data_structure_custom.o \
        fft_parallel.o \
        fft_parallel_gpu.o \
        fft_interfaces.o \
        fft_interfaces_gpu.o \
        wrappers.o \
        funct.o \
        griddim.o \
        image_io_routines.o \
        allocate_fft_custom.o \
        ruotaijk.o \
        xk_wk_collect.o \
        mp_base.o \
        mp_wave.o \
        mp_image_global_module.o \
        pseudo_types.o \
        zhpev_drv.o \
        ptoolkit.o \
        read_cards.o \
        read_namelists.o \
        uspp.o \
        upf_nml.o \
        upf.o \
        upf_to_internal.o \
        read_pseudo.o \
        recvec_subs.o \
        run_info.o \
        set_signal.o \
        splinelib.o \
        wavefunctions.o \
        \
        c_mkdir.o \
        cptimer.o \
        customize_signals.o \
        eval_infix.o \
        fft_stick.o \
        md5.o \
        md5_from_file.o \
        memstat.o \
        stack.o \
        \
        atomic_number.o \
        capital.o \
        cryst_to_car.o \
        date_and_tim.o \
        distools.o \
        erf.o \
        find_free_unit.o \
        flush_unit.o \
        functionals.o \
        inpfile.o \
        int_to_char.o \
        invmat.o \
        latgen.o \
        lsda_functionals.o \
        matches.o \
        recips.o \
        remove_tot_torque.o \
        rgen.o \
        simpsn.o \
        sort.o \
        sph_bes.o \
        trimcheck.o \
        volume.o \
        ylmr2.o \
        \
        symm_base.o \
        start_k.o \
        scf_mod.o \
        a2fmod.o \
        buffers.o \
        becmod.o \
        add_vuspsi.o \
        allocate_fft.o \
        allocate_locpot.o \
        allocate_nlpot.o \
        allocate_wfc.o \
        atomic_rho.o \
        atomic_wfc.o \
        g_psi_mod.o \
        c_bands.o \
        ccgdiagg.o \
        cdiaghg.o \
        cdiaghg_gpu.o \
        cegterg.o \
        symme.o \
        close_files.o \
        coset.o \
        data_structure.o \
        deriv_drhoc.o \
        divide.o \
        divide_et_impera.o \
        drhoc.o \
        dvloc_of_g.o \
        compute_deff.o \
        newd.o \
        coulomb_vcut.o \
        exx.o \
        clean_pw.o \
        input.o \
        electrons.o \
        eqvect.o \
        ewald.o \
        g2_kin.o \
        g_psi.o \
        gk_sort.o \
        gradcorr.o \
        h_1psi.o \
        h_psi.o \
        hinit0.o \
        init_at_1.o \
        openfil.o \
        init_run.o \
        init_us_1.o \
        init_us_2.o \
        init_vloc.o \
        interpolate.o \
        irrek.o \
        iweights.o \
        kpoint_grid.o \
        lchk_tauxk.o \
        memory_report.o \
        mix_rho.o \
        multable.o \
        n_plane_waves.o \
        para.o \
        potinit.o \
        print_clock_pw.o \
        print_ks_energies.o \
        read_input.o \
        pwscf.o \
        remove_atomic_rho.o \
        rotate_wfc.o \
        rotate_wfc_k.o \
        s_1psi.o \
        s_psi.o \
        set_kup_and_kdw.o \
        set_rhoc.o \
        set_vrs.o \
        setlocal.o \
        setup.o \
        stop_run.o \
        struct_fact.o \
        sum_band.o \
        summary.o \
        usnldiag.o \
        v_of_rho.o \
        vloc_of_g.o \
        vloc_psi.o \
        vloc_psi_gpu.o \
        weights.o \
        wfcinit.o

# rules
all: mini_dft $(if $(USE_HPCTK), mini_dft.hpcstruct)

%.o : %.f90
        $(FC) $(FFLAGS) -c $<

```

```
mini_dft: $(OBJECTS)
	$(LD) $(LDFLAGS) -o $@ fortran_thunking.o $(OBJECTS) $(LIBS) $(IPM)
	-rm -f mini_dft.hpcstruct

mini_dft.hpcstruct: mini_dft
	hpcstruct $<

.PHONY = clean
clean:
	-rm -f *~ *.o *.mod mini_dft

.PHONY = tarball
tarball:
	tar -czf mini_dft.tar.gz Makefile *.UPF *.UPF.nml *.h *.c *.f *.f90
```

After successful compilation, launch the program with MPI and the run script in Section 1.3.2 (adjust according to your environment). The number of MPI ranks should equal the number of CPU cores. For example,

```
$ mpirun -np 88 -ppn 44 -hosts compute0,compute1 bash run.sh
```

2.1 Fortran thunking interface

Nvidia provides the Fortran thunking cuBLAS interface. It's available in `/path/to/cuda/install/dir/src`. To compile it, do

```
$ nvcc -O3 -c fortran_thunking.c -Xcompiler -DCUBLAS_GFORTRAN
```

2.2 Nvidia MPS

To run Nvidia MPS, first set GPU to `EXCLUSIVE_PROCESS` mode. Then start the MPS server as root.

```
$ sudo nvidia-smi -c 3
$ sudo nvidia-cuda-mps-control -d
```

3 Result

We performed extensive benchmarking on our cluster for miniDFT. The specification of our nodes is in Table 1, and we have two identical nodes in total. The runtime we recorded for original and final versions are in Table 2. No command line option besides `-in pe-23.LOCAL.in` is used. The run script in Section 1.3.2 is used for launching jobs for the final version.

As shown in Table 2, we were able to achieve a speed up of 3.1 on our cluster for the input `pe-23.LOCAL.in`.

Table 1: Machine specification

Item	**Count**
E5-2699 v4 (2.2GHz, 22 cores) CPU	2
DDR4 16GB 2400MHz RAM	16 (256GB total)
Mellanox InfiniBand EDR adapter	1
Nvidia Tesla P100 accelerator	4
SAMSUNG 500GB SSD	1

Table 2: Runtime

Version	**Benchmark Time**
Original (only compilation optimization)	504.36 s
Final (comp ops, algo ops, cuBLAS, MAGMA, proc binding)	162.60 s