OpenACC

Parallel Programming
2025/10/30

國立清華大學叢集電腦競賽團隊
Student Cluster Competition Team of NTHU

What is OpenACC?

- **Open Acc**elerators
- Through various **compiler directives** to write GPU code
- Lower the technical barriers to GPU programming

What is OpenACC

```
#pragma acc data copy(A) create(Anew)
while ( error > tol  &&  iter  <  iter_max )  {
  error = 0.0;
#pragma acc kernels
  {
#pragma acc loop independent collapse(2) reduction(max:error)
  for (  int  j = 1; j < n-1;  j++ )  {
    for (  int  i = 1; i < m-1; i++ )  {
       Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                               A [j-1] [i] + A [j+1] [i]);
       error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
      }
    }
    ...
  }
}
```

https://www.openacc.org/

CUDA

- CudaMalloc(...): Declare memory on the GPU
- CudaMemcpy(...): Move data
- functionname<<<thread, blocks>>>(...): Write your own Cuda Kernel Function

=> High entry barrier

CUDA Example

```c
#define N 10000000
int main() {
    float x[N], y[N];
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

Port to GPU

```c
#define N 10000000
// CUDA kernel function
__global__ void cuda_sin(float *x, float *y, int n) {
    int threadId = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadId >= n) {
        return;
    }
    y[threadId] = sin(x[threadId]);
}
int main() {
    float x[N], y[N];
    // 宣告 GPU device 上的 array
    float *x_device, *y_device;
    cudaMalloc(&x_device, sizeof(float) * N);
    cudaMalloc(&y_device, sizeof(float) * N);
    // 將 x 陣列複製到 GPU 上
    cudaMemcpy(x_device, x, sizeof(float) * N, cudaMemcpyHostToDevice);
    // 計算所需要的 thread block 數量
    int thread_block_sz = 1024;
    int num_blocks = N / thread_block_sz;
    if (N % thread_block_sz != 0) {
        num_blocks += 1;
    }
    // 呼叫 CUDA kernel function
    cuda_sin<<<num_of_blocks, thread_block_sz>>>(x_device, y_device, N);
    // 把結果複製回 y 陣列
    cudaMemcpy(y, y_device, sizeof(float) * N, cudaMemcpyDeviceToHost);
    // 釋放 GPU 記憶體
    cudaFree(x_device);
    cudaFree(y_device);
}
```

OpenACC

- No need to declare the memory on the device
- #pragma acc data copy(...): You can move data with a simple clause
- You can directly use parallel region to port to the GPU.

=> Easy to use

```
#define N 10000000
int main() {
    float x[N], y[N];
    #pragma acc data copyin(x[0:N]) copyout(y[0:N])
    #pragma acc parallel loop
    for (int i = 0; i < N; i++) {
        y[i] = sin(x[i]);
    }
}
```

OpenACC Directive

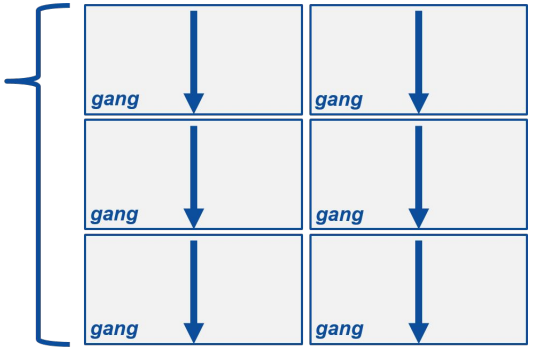
- #pragma acc <directive> <clause>
  - #pragma is a compiler hint
  - acc tells the compiler that this is the OpenACC pragma
  - directive is what OpenACC tells the compiler to indicate
  - clause is an instruction for OpenACC to supplement or optimize the directive.
  - will take effect on next block/line.

OpenACC Directive - parallel

- #pragma acc parallel
  - parallel tells the compiler that this code should be **redundantly parallelized**
  - tells the compiler to run this part of code on **GPU**

```
#pragma acc parallel
{

}
```

When encountering the ***parallel*** directive, the compiler will generate *1 or more parallel* ***gangs***, which execute redundantly.

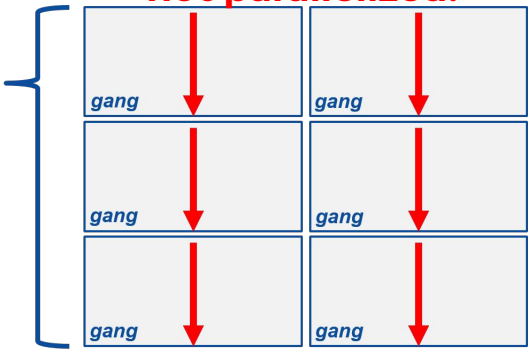



```
#pragma acc parallel
{

    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

}
```

This loop will be ***redundantly parallelized*** across the ***gangs***

This means that each ***gang*** will execute the entire loop

not parallelized!



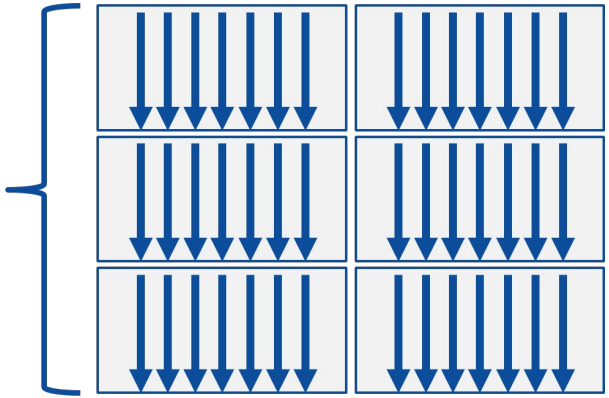
OpenACC Directive - loop

- #pragma acc parallel loop
  - loop tells the compiler that this loop needs to be parallelized
  - It also tells the compiler that this loop can be safely parallelized.

```
#pragma acc parallel
{

    #pragma acc loop
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

}
```

The ***loop*** directive informs the compiler which loops to parallelize.

The iterations of the loop will be broken up evenly among the parallel ***gangs***.



The ***gangs*** will then execute in parallel with one another.

OpenACC Directive - reduction

- #pragma acc parallel loop reduction(<operation>:<target>)
  - reduction tells the compiler that a target is to be reduced
  - reduce: perform global operations on the selected target

```
int sum = 0;
#pragma acc parallel loop reduction(+:sum)
for(int i = 0, i < N, i++) sum += i;
```

reduction(+ : sum)
reduction(* : prod)
reduction(min : mn)
reduction(& : mask)

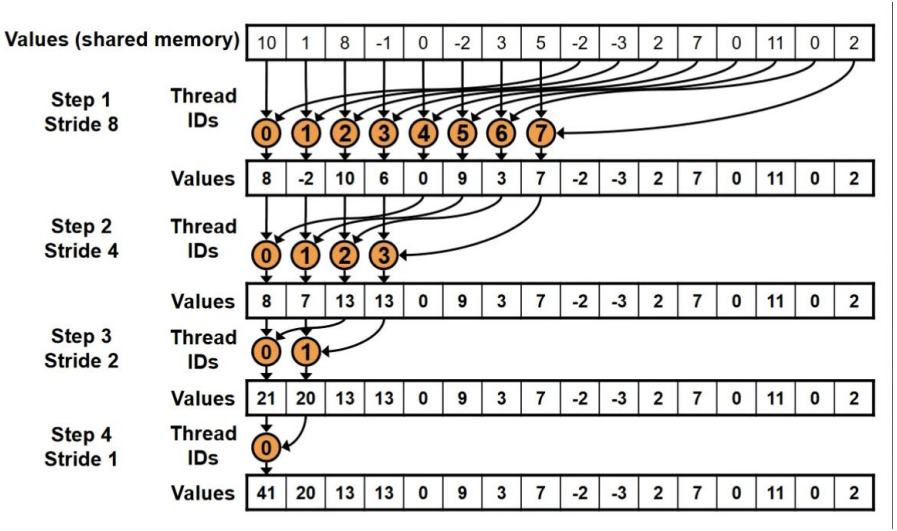
OpenACC Directive - reduction cont'd

- How reduction work?
  - Each gang computes its own local result in parallel.
  - After the loop ends, OpenACC reduces all gang results into a single final value.

```
int main() {
    float sum = 0;
    float mn = 999999;
    float x[N];
    #pragma acc parallel loop reduction(min:mn) reduction(+:sum)
    for (int i = 0; i < N; i++) {
        float tmp = sin(x[i]);
        sum = sum + tmp;
        mn = min(mn, tmp);
    }
}
```

OpenACC Directive - parallel reduction

- Works only when there's no data dependency between iterations.
- Reduces in a tree-like fashion — pairs of partial results are combined.
- Overall time complexity becomes O(log N). (N is the number of gangs)

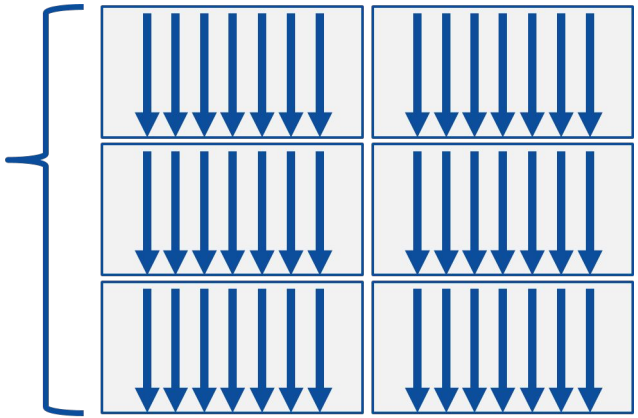


OpenACC Directive - kernels

- #pragma acc kernels
  - All actions are decided by the compiler
  - You can also include the sequential code

```
#pragma acc kernels
{
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

    for(int i = 0; i < M; i++)
    {
        // Do Something Else
    }
}
```

With the ***kernels*** directive, the ***loop*** directive is implied.



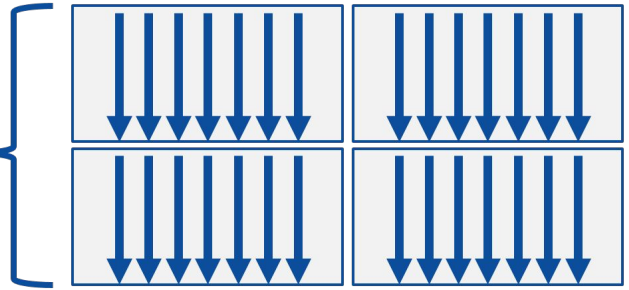
OpenACC Directive - kernels

- #pragma acc kernels loop independent
  - Tell the compiler that this loop can be safely parallelized, and force parallelization of it

```
#pragma acc kernels
{

    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

    for(int i = 0; i < M; i++)
    {
        // Do Something Else
    }
}
```

This process can happen multiple times within the ***kernels*** region.

Each loop can have a different number of gangs, and those gangs can be organized/optimized completely differently.



```
#pragma acc kernels
{
    for (int i = 0; i < N; i++)
    {
        < loop code >
    }

    < some other sequential code >

    #pragma acc loop independent
    for (int j = 0; j < M; j++)
    {
        < loop code >
    }
}
```

Data Management

```
#pragma acc data copy(A[0:N])
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; i < N; i++) A[i] = 0;
}
```


```
#pragma acc kernels copy(a[0:N])
for(int i = 0; i < N; i++){
  a[i] = 0;
}
```

Data Management

You can copy only part of the data
#pragma acc parallel loop copy(A[1:N-2])

- #pragma acc data copy(...)
  - Copy the data into the GPU and copy the data back to the CPU after the parallel region ends
- #pragma acc data copyin(...)
  - Copy the data into the GPU and delete the data on the GPU after the parallel region ends.
- #pragma acc data copyout(...)
  - Copy the data back to the CPU and delete the data on the GPU after the parallel region ends.
- #pragma acc data create(...)
  - Declare a space on the GPU without performing any copying operations
  - When there are variables for temporary storage, using this clause eliminates the need to copy in and out.

Loop Optimization

- #pragma acc parallel loop collapse(...)
  - Can be used in tightly nested loops
  - collapse can flatten loops and turn multiple loops into one large parallel loop

```
#pragma acc parallel loop collapse( 2 )
for(int j = 0; j < M; j++) {
    for(int k = 0; k < Q; k++) {
        < loop code >
    }
}
```

TIP1:
When the outer loop is too small, flattening the loop can increase GPU usage.

Loop Optimization

- #pragma acc parallel loop tile(x, y)
  - Calculate nested loop break for multiple tiles (blocks)

```
#pragma acc parallel loop tile( 32, 32 )
for(int j = 0; j < 128; j++) {
    for(int k = 0; k < 128; k++) {
        < loop code >
    }
}
```

TIP1:
Try to make the tile size a multiple of 32. The threads in a worker and vector of Nvidia GPU are executed in units of 32.

TIP2:
Do not use tiles larger than 32*32, because in NVIDIA GPU, the maximum number of threads in a gang is 1024 (32*32)

Practice

Try to parallelize this code.

```
#include <bits/stdc++.h>
#define N 100000
double x[N], y[N], sum = 0;
void generate_random() {
    srand(0);
    for (int i = 0; i < N; i++) {
        x[i] = (rand() / (double)RAND_MAX) * 10.0 - 5.0;
    }
}

int main() {
    generate_random();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            y[i] += x[i] * x[j] + cos(x[i] + x[j]);
            sum = sum + y[i];
        }
    }
    std::cout << sum << '\n';
}
```

Execution time > 30 seconds

How to compile

nvc++ -fast -acc -gpu=cc61 -Minfo=accel -o <執□檔名稱> <檔案.cpp>

- **-acc** tell compiler to support OpenACC
- **-gpu=cc61** assign GPU architecture GTX1080(cc61), V100(cc70)
- **-Minfo=accel** print info about the code is parallelized

```
gilbert12@apollo-login:~$ nvc++ -fast -acc -gpu=cc61 -Minfo=accel -o acc acc.cpp
main:
     12, Generating copyin(x[:]) [if not already present]
         Generating NVIDIA GPU code
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:sum)
         16, #pragma acc loop seq
     12, Generating copyout(y[:]) [if not already present]
         Generating implicit copy(sum) [if not already present]
     16, Complex loop carried dependence of y prevents parallelization
         Loop carried dependence of y prevents parallelization
         Loop carried backward dependence of y prevents vectorization
```

Answer

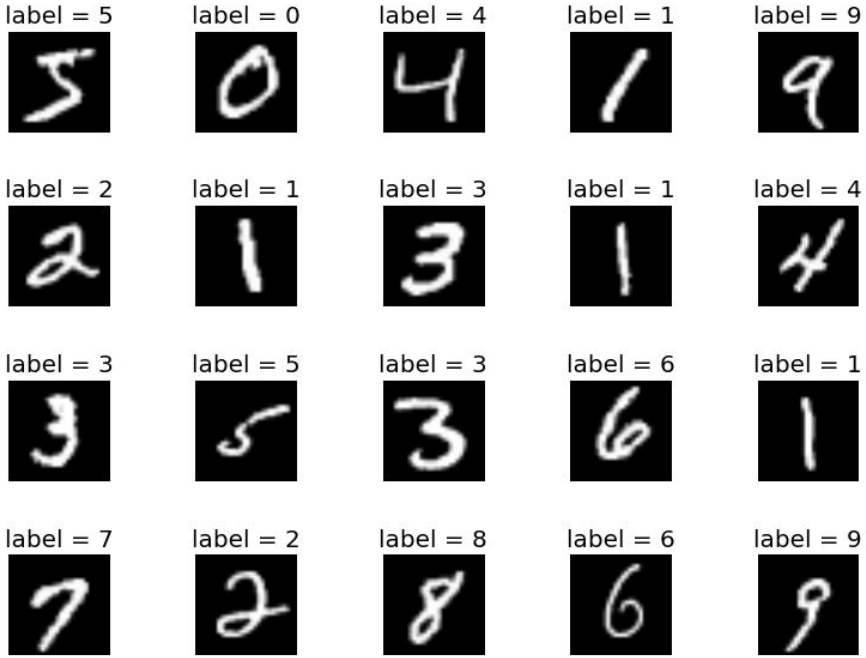
```
#pragma acc data copyin(x[0:N]) copyout(y[0:N])
#pragma acc parallel loop reduction(+:sum)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        y[i] += x[i] * x[j] + cos(x[i] + x[j]);
        sum = sum + y[i];
    }
}
std::cout << sum << '\n';
```

Execution time < 1 second

OpenACC Labs

Deep Neural Network(DNN)

- MNIST hand written digits classification Inference



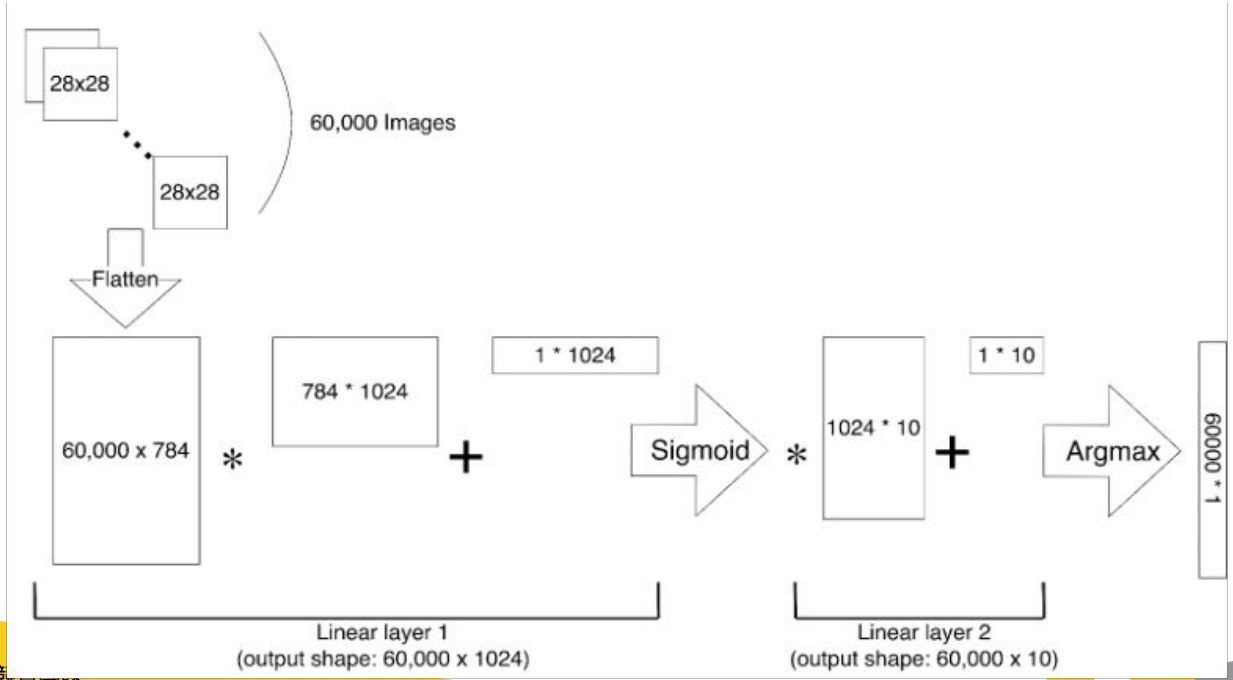
Deep Neural Network(DNN)

- Two fully-connected layer

```
simpleNN(
  (nn): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=784, out_features=1024, bias=True)
    (2): Sigmoid()
    (3): Linear(in_features=1024, out_features=10, bias=True)
    (4): Softmax(dim=None)
  )
)
```

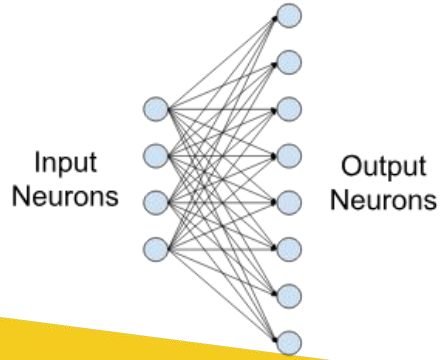
Deep Neural Network(DNN)

- Two fully-connected layer

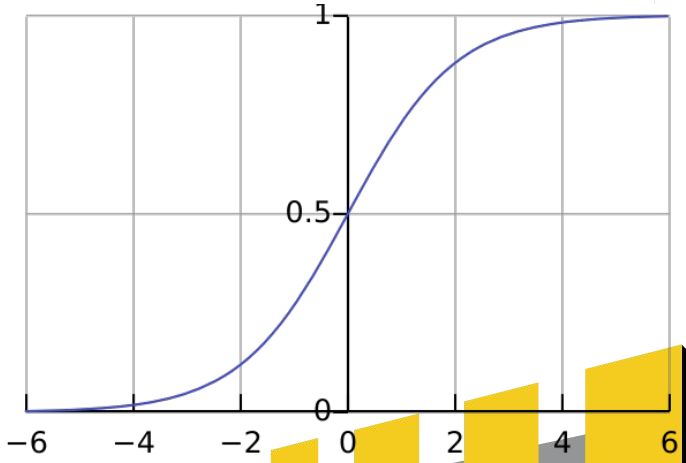


Deep Neural Network(DNN)

- Two fully-connected layer
  - First Linear Layer + Sigmoid Activation Function
  - Second Linear Layer + Argmax



$$\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$



Task

- Given the sequential version of DNN code that runs on CPU.
- Try to parallel the matrix computation in the neural network.
- Using OpenACC

Task

- Provided files:
  - Sequential code
  - pretrained model weights(since the task is inference)
  - Makefile
- Please only modify the functions with TODO label.

Workflow

1. cp -r /home/pp25/share/lab-mnist $HOME
2. cd ~/lab-mnist
3. Parallel the TODOs in mnist.cpp
4. module load nvhpc-nompi
5. compile the program : make
6. run the DNN program : srun -p nvidia --gres=gpu:1 ./mnist
7. judge: lab-mnist-judge
   scoreboard: lab-mnist

Result

- Inference accuracy should be **97.8183%** (the parallelization should not affect the accuracy)

Result

```
gilbert12@apollo-login:~/lab-mnist$ srun -p nvidia mnist
srun: No GPU specified, defaulting to allocating 1 GPU
CUDA initialized.
Nbr of training images = 60000
Reading file: /share/testcases/lab-mnist/weights/layer1_matrix
Reading file: /share/testcases/lab-mnist/weights/layer1_bias
Reading file: /share/testcases/lab-mnist/weights/layer2_matrix
Reading file: /share/testcases/lab-mnist/weights/layer2_bias

Inference accuracy: 97.8183%

-----     STATS     -----
Time for initializing CUDA device:     104 m.s.
Time for reading MNIST data & weights: 173 m.s.
Time for inferencing                 : 545 m.s.
Time for calculating accuracy        : 6 m.s.
-----  END OF STATS  -----
```

Judge Result

- Judge: lab-mnist-judge
- Scoreboard: lab-mnist
- Platform: apollo-gpu