南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计期末实验开题报告

英特尔 oneAPI—高斯消元算法并行化

蒋薇

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2023 年 5 月 14 日

摘要

以高斯消去为解决问题的实际目的，借助 OneAPI 工具作为载体，从基本编程方面、代码编写方面、辅助库方面、分析调试方面对 oneAPI 与传统工具进行对比研究，再 j

关键字：Parallel,oneAPI,Gaussian

目录

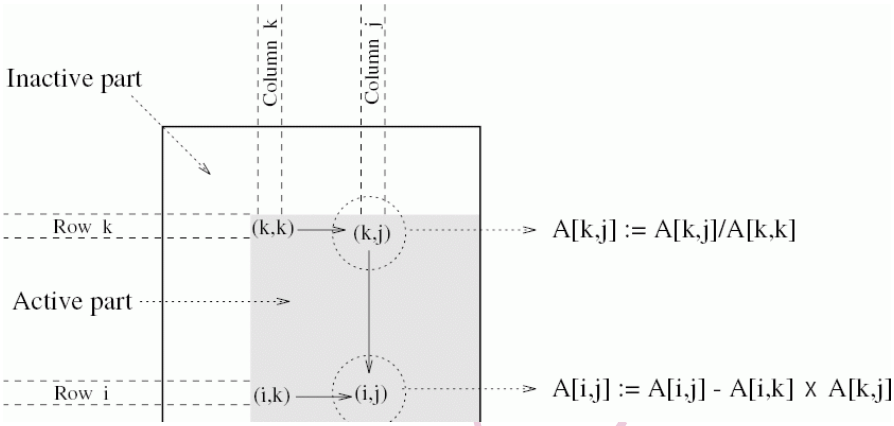
一、 英特尔 oneAPI—高斯消元算法并行化横向对比研究

(一) 问题描述

1. 高斯消元法

高斯消元法 (Gaussian elimination) 是求解线性方阵组的一种算法，它也可用来求矩阵的秩，以及求可逆方阵的逆矩阵。它通过逐步消除未知数来将原始线性系统转化为另一个更简单的等价的系统。它的实质是通过初等行变化 (Elementary row operations)，将线性方程组的增广矩阵转化为行阶梯矩阵。

(二) 算法分析



高斯消去的计算模式如上图所示，在第 k 步时，首先将第 k 行除以 A[k,k]，此时 A[k,k] = 1，然后再将第 k 行到第 n 行减去第一行乘以 A[i,k]，此时位于 A[k,k] 下方的所有元素全部为 0。总共进行 n 步操作后，矩阵 A 就变成了对角线元素全为 1 的上三角矩阵。

算法伪代码如下:

高斯消元

```
procedure LU (A)
begin
    for k := 1 to n do
        for j := k+1 to n do
                A[k, j ] := A[k, j]/A[k, k];
        endfor;
        A[k, k] := 1.0;
        for i := k + 1 to n do
                for j := k + 1 to n do
                A[i, j ] := A[i, j ] - A[i, k]×A[k, j];
            endfor;
                A[i, k] := 0;
        endfor;
    endfor;
end LU
```

(三) 算法设计

1. 串行算法

具体代码如下:

高斯消元

```
procedure LU (A)
befor (int k = 0; k < n; k++) {
        for (int j = k + 1; j < n; j++) {
                m[k][j] = m[k][j] / m[k][k];
        }
        m[k][k] = 1;
        for (int i = k + 1; i < n; i++) {
                for (int j = k + 1; j < n; j++) {
                        m[i][j] = m[i][j] - m[i][k] * m[k][j];
                }
                m[i][k] = 0;
        }
}
```

2. DPC++ 并行高斯消元

DPC++ 语言实现代码在不同架构间的复用，DPC++ 编译器实现针对不同架构的目标代码调优，在实现统一编程便捷性的同时又不牺牲性能。DPC++ Compatibility tool 可帮助迁移用 CUDA 编写的现有代码到 DPC++ 程序，可实现 80-90% 代码的自动迁移，提供行内注释帮助程序员完成剩余迁移。DPC++ 库：C++17 算法的优化、基于并行 STL、Boost 等著名的并行库、集成兼容工具。

textbf 并行化的循环层数

三重循环，其中第一重循环遍历矩阵所有行，当前选中行（第 i 行）便被用来消去其他行；第二重循环被用来遍历第 i 行“下面”的所有矩阵行（第 j 行）；而第三重循环则被用来将第 j 行减去第 i 行，来实现将位于 (j, i) 位置的矩阵元素置零的作用。

第一重循环存在数据依赖——执行顺序在后的循环步需要执行顺序在前的循环步提供数据，不好做并行化；第三重循环很容易并行化，但是矩阵一行数千个元素，进行任务划分和数据发送的时间可能都比计算要花的时间长；因此，我们的并行化应该做在第二重循环上。代码如下:

DPC++ 高斯消元

```
queue myQueue{ host_selector{} }; //创建队列
for (int i = 0; i < n; i++)
{
        myQueue.parallel_for(range{(unsigned long)(n - (i + 1))}, [=](id<1>
            idx)
        {
            int j = idx[0] + i + 1; // 等同于for(int j=i+1; j<n; j++)
            float div = new_mat[j][i] / new_mat[i][i];

            for (int k = i; k < n; k++)
            {
```

```
                new_mat[j][k] -= new_mat[i][k] * div;
            }
        }).wait(); // wait() - 等待所有任务执行完成
}
```

代码简洁优雅

queuemyQueuecpu_selector;// 创建一个队列（queue），在 OneAPI 中，队列是用来沟通主机（host）和运算设备（devices）的一种途径，我们可以通过向队列提交任务来让他们在运算设备（devices）上运行。初始化队列使用到的 cpu_selector，可以指定这个队列的运算设备为 CPU，接下来的算法也就自然会在 CPU 上运行。

GPU *queuemyQueuegpu_selector*;// 创建队列时，将 cpu_selector 改为 gpu_selector 可以通过 gpu 运算，不过我们的代码在 gpu 上运行的速度不怎样快。

myQueue.parallel_for(*rangen*, [=](*id* < 1 > *idx*)*do_something*;).*wait*(); parallel_for 函数正如其名，是并行化的 for 语句。parallel_for 接受两个参数，第一个参数属于 range 类，比如这里的 rangen 便表示循环范围是 [0, n)，后面的 idx 便是循环变量。而第二个参数则是一个 lambda 匿名函数, 函数体内写我们要实现的功能; 最后调用的 wait() 函数是为了让程序阻塞, 等待当前所有任务运行完毕后再开始下一次循环，避免出错。

intj = *idx*[0] + *i* + 1; 因为 parallel_for 只能支持循环变量初值为 0 的情况的，所以需要我们手动加上一个偏移量。

oneAPI 高斯消元代码 请点击这里

3. SIMD 并行高斯消元

4. 多核 CPU 并行高斯消元

5. GPU 高斯并行消元

6. 基于 buffer 并行高斯消元

首先创建一个任务队列，用于给加速器提交任务：*queueq*;

在创建队列时, 将队列绑定到不同设备上, 例如: queue(cpu_selector); queue(gpu_selector); 指定我们的并行代码要在什么设备上执行，不需要了解 GPU/FPGA 等加速器具体的编程语言，就可以将计算任务以统一的编程方式卸载到这些加速器上。

然后创建一个 n*n 大小的矩阵:

const int n = 1024; buffer<float, 2> buf(range(n, n));

针对第一个部分，使用 parallel_for 语句将其卸载到加速器上并行计算，具体代码如下:

基于 buffer 并行高斯消元

```
q.submit([&](handler& h) {
        accessor m{ buf, h, read_write };
        h.parallel_for(range(n - k), [=](auto idx) {
                int j = k + idx;
                m[k][j] = m[k][j] / m[k][k];
        });
});
```

针对第二个部分，我们仍然可以使用 parallel_for，注意，这里使用的不再是一维的数据划分，而是二维的数据划分，具体代码如下:

基于 buffer 并行高斯消元

```
q.submit([&](handler& h) {
        accessor m{ buf, h, read_write };
        h.parallel_for(range(n - (k + 1), n - (k + 1)), [=](auto idx) {
                int i = k + 1 + idx.get_id(0);
                int j = k + 1 + idx.get_id(1);
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
        });
});
```

注意，由于 A[i,k] 元素在此部分中会被所有线程使用，因此不能在此阶段置 0，需要等这部分全部执行完成之后再置 0。因此需要补充第三个阶段，将右下角 k*k 大小的矩阵的第一列除 A[k,k] 以外的元素置 0，具体代码如下：

基于 buffer 并行高斯消元

```
q.submit([&](handler& h) {
        accessor m{ buf, h, read_write };
        h.parallel_for(range(n - (k + 1)), [=](auto idx) {
                int i = k + 1 + idx;
                m[i][k] = 0;
        });
});
```

在最外层 n 重循环执行完之后，其实只是将计算任务下发到了加速器，这些计算任务不一定真的执行完了，需要手动调用 q.wait()，等待任务队列中所有任务都执行完毕，该算法再结束。

完整代码如下：

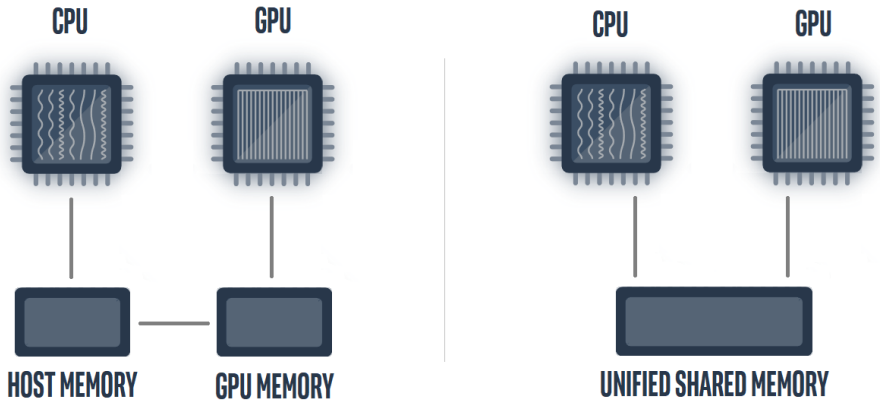
基于 buffer 并行高斯消元

```
void gauss_oneapi(buffer<float, 2>& buf, queue& q) {
        device my_device = q.get_device();
        std::cout << "Device: " << my_device.get_info<info::device::name>()
            << std::endl;

        int n = buf.get_range()[0];
        for (int k = 0; k < n; k++) {
                q.submit([&](handler& h) {
                        accessor m{ buf, h, read_write };
                        h.parallel_for(range(n - k), [=](auto idx) {
                                int j = k + idx;
                                m[k][j] = m[k][j] / m[k][k];
                        });
                });

                q.submit([&](handler& h) {
                        accessor m{ buf, h, read_write };
                        h.parallel_for(range(n - (k + 1), n - (k + 1)), [=](
                            auto idx) {
                                int i = k + 1 + idx.get_id(0);
```

```
                int j = k + 1 + idx.get_id(1);
                m[i][j] = m[i][j] - m[i][k] * m[k][j];
            });
        });

        q.submit([&](handler& h) {
            accessor m{ buf, h, read_write };
            h.parallel_for(range(n - (k + 1)), [=](auto idx) {
                int i = k + 1 + idx;
                m[i][k] = 0;
            });
        });
    }
    q.wait();
}
```

7. 基于 USM 并行高斯消元

USM 的全称是 Unified Shared Memory，程序员可以不管数据是处在主机的内存当中还是加速器的内存当中，仍然使用 C/C++ 中的指针来访问数据，它给程序员提供了一个统一的内存视图，有助于程序员更快地将现有 C/C++ 程序移植到 DPC++。



注意：和 buffer+accessor 的访问方式相比，如果对数据依赖解决不正确，可能会存在竞争，导致计算结果错误。

在申请矩阵内存时，我们可以使用 malloc_shared 函数：

$float * buf = (float*)malloc_shared(n * n * sizeof(float), q);$

在访问内存时，则不需要使用 accessor，直接访问 buf 即可。需要注意由于不再使用 accessor，因此 oneAPI 不能推断出数据依赖，所以需要程序员手动指出数据依赖，其中有 3 种方式：

q.wait()

in_order 队列属性

h.depends_on(e) 方法

本算法中所有 kernel 都是依次执行的，因此在创建队列时指定 in_order 属性比较方便。

$queueqproperty :: queue :: in_order();$

具体算法如下:

基于 USM 并行高斯消元

```
void gauss_oneapi(float* m, int n, queue& q) {
        device my_device = q.get_device();
        std::cout << "Device: " << my_device.get_info<info::device::name>()
            << std::endl;

        for (int k = 0; k < n; k++) {
                q.submit([&](handler& h) {
                        h.parallel_for(range(n - k), [=](auto idx) {
                                int j = k + idx;
                                m[k*n+j] = m[k * n + j] / m[k * n + k];
                        });
                });

                q.submit([&](handler& h) {
                        h.parallel_for(range(n - (k + 1), n - (k + 1)), [=](
                            auto idx) {
                                int i = k + 1 + idx.get_id(0);
                                int j = k + 1 + idx.get_id(1);
                                m[i * n + j] = m[i * n + j] - m[i * n + k] *
                                    m[k * n + j];
                        });
                });

                q.submit([&](handler& h) {
                        h.parallel_for(range(n - (k + 1)), [=](auto idx) {
                                int i = k + 1 + idx;
                                m[i * n + k] = 0;
                        });
                });
        }
        q.wait();
}
```

barrierUSM 代码 请点击这里

(四) 实验设计

准备 时间测量方法: 我们使用 C++11 标准中的 *std* :: *chrono* :: *high_resolution_clock* 测量时间，而不使用平台特定的 API(如 Windows 平台的 QueryPerformanceCounter、Linux 平台的 clock 等)，这样可以增加程序的可移植性。

除此之外，还应使用多次测量求平均的方式保证运行时间测量的准确性。

auto start = std::chrono::high_resolution_clock::now();

// my code

auto end = std::chrono::high_resolution_clock::now();

预热: 为了防止初始化操作、cache 等其他因素影响算法执行时间的测量，因此在开始计时之前，先让算法运行一次，第一次不计入运行时间，然后再运行 k 次，以这 k 次运行的平均时间作为算法执行时间。

测试环境:

性能测试部分全部在 DevCloud 平台上完成。

GPU 型号：Intel(R) UHD Graphics P630

CPU 型号：Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz

1. 串行算法与基于 buffer 实现的并行算法的性能对比

为了对比串行算法与并行算法的性能差异，设置了不同规模的矩阵，分别测量在不同问题规模下，串行算法与并行算法的性能表现（单位：ms）。

问题规模 n	串行算法	GPU 并行算法
16	0.470433	1.42439
32	0.812061	2.292
64	6.41777	4.03059
128	50.598	5.60625
256	406.665	7.42548
512	3236.67	75.7885
1024	25885.7	590.462

表 1: 串行算法与基于 buffer 实现的并行算法的算法运行时间对比

当问题规模 n 非常小的情况下，串行算法的效率是最高的。因为如果我们使用 GPU 做并行化，那么就需要将矩阵从主机的内存复制到 GPU 的内存中，然后再让 GPU 开始计算，等 GPU 计算完之后，还要把矩阵再从 GPU 的内存复制到主机的内存中。在问题规模 n 非常小的情况下，这些额外的开销占比是非常大的，因此在此时 GPU 版本算法要比串行算法的执行时间更长。

当问题规模 n 比较大时，GPU 并行算法耗时更短，比较符合预期效果。

2. 基于 buffer 实现的并行算法与基于 USM 实现的并行算法的性能对比

为了对比基于 buffer 实现的并行算法与基于 USM 实现的并行算法之间的性能差异，也应测量在不同问题规模下各算法的性能表现。

问题规模 n	GPU 并行算法（buffer）	GPU 并行算法（USM)）	
16	1.42439	1.59059	
32	2.292	2.02017	
64		4.03059	2.9549
128	5.60625	4.81008	
256	7.42548	5.91026	
512	75.7885	89.6216	
1024	590.462	616.103	

表 2: 基于 buffer 实现的并行算法与基于 USM 实现的并行算法的运行时间对比

由上表可以看出，基于 buffer 的 GPU 并行算法与基于 USM 的 GPU 并行算法之间性能差异不大，因此程序员可以使用 USM 更快地将现有 C/C++ 程序移植到 DPC++，并且相比于使用 buffer+accessor，不会带来很多额外的性能开销。

3. **SIMD 并行架构下 SSE/AVX、Pthread、OpenMP、CUDA 与 DPC++ 并行高斯消元对比**

   SIMD 并行架构下 SSE/AVX 实现高斯消元
   SIMD 并行架构下 Pthread 实现高斯消元
   SIMD 并行架构下 OpenMP 实现高斯消元
   SIMD 并行架构下 CUDA 实现高斯消元
   SIMD 架构下不同工具与 oneAPI 下 DPC++ 高斯消元性能对比

4. **多核 CPU 并行架构下 SSE/AVX、Pthread、OpenMP、CUDA 与 DPC++ 并行高斯消元对比**

   多核 CPU 并行架构下 SSE/AVX 实现高斯消元
   多核 CPU 并行架构下 Pthread 实现高斯消元
   多核 CPU 并行架构下 OpenMP 实现高斯消元
   多核 CPU 并行架构下 CUDA 实现高斯消元
   多核 CPU 架构下不同工具与 oneAPI 下 DPC++ 高斯消元性能对比

5. **GPU 并行架构下 SSE/AVX、Pthread、OpenMP、CUDA 与 DPC++ 并行高斯消元对比**

   GPU 并行架构下 SSE/AVX 实现高斯消元
   GPU 并行架构下 Pthread 实现高斯消元
   GPU 并行架构下 OpenMP 实现高斯消元
   GPU 并行架构下 CUDA 实现高斯消元
   GPU 架构下不同工具与 oneAPI 下 DPC++ 高斯消元性能对比

6. **DPC++ Compatibility tool 对 CUDA 程序进行到 DPC++ 程序的迁移与 CUDA 程序对比**

   DPC++ Compatibility tool 对 CUDA 程序进行到 DPC++ 程序的迁移的具体实现
   CUDA 程序
   两者对高斯消元性能影响的对比

7. **DPC++ Compatibility tool 对 CUDA 程序进行到 DPC++ 程序的迁移与 DPC++ 程序对比**

   DPC++ Compatibility tool 对 CUDA 程序进行到 DPC++ 程序的迁移以及 DPC++ 程序上文已实现，**两者对高斯消元性能影响的对比**

8. **DPC++ Compatibility tool 对 CUDA 程序进行到 DPC++ 程序的迁移与采用 Compatibility tool 迁移的 DPC++ 程序对比**

   采用 Compatibility tool 迁移的 DPC++ 程序
   两者对高斯消元性能影响的对比

9. Advisor 程序优化对高斯消元性能的影响

Advisor 简介:
Advisor 具体实现高斯消元:
使用 Advisor 与未使用对程序性能的影响

10. DPC++ 库与不借助库在编程高斯消元的对比

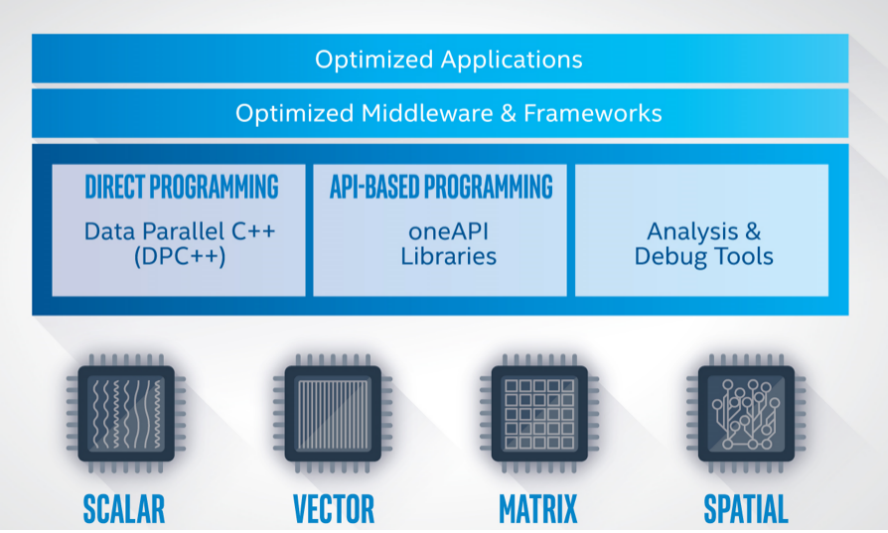
利用 DPC++ 库编写高斯消去程序
不借助库编写高斯消去程序
编程工作量、目标程序性能对比
分析优化 DPC++ 程序
使用 Vtune 分析 DPC++ 程序
寻找性能瓶颈
程序相应优化
Vtune 再分析验证效果

二、 纵向研究

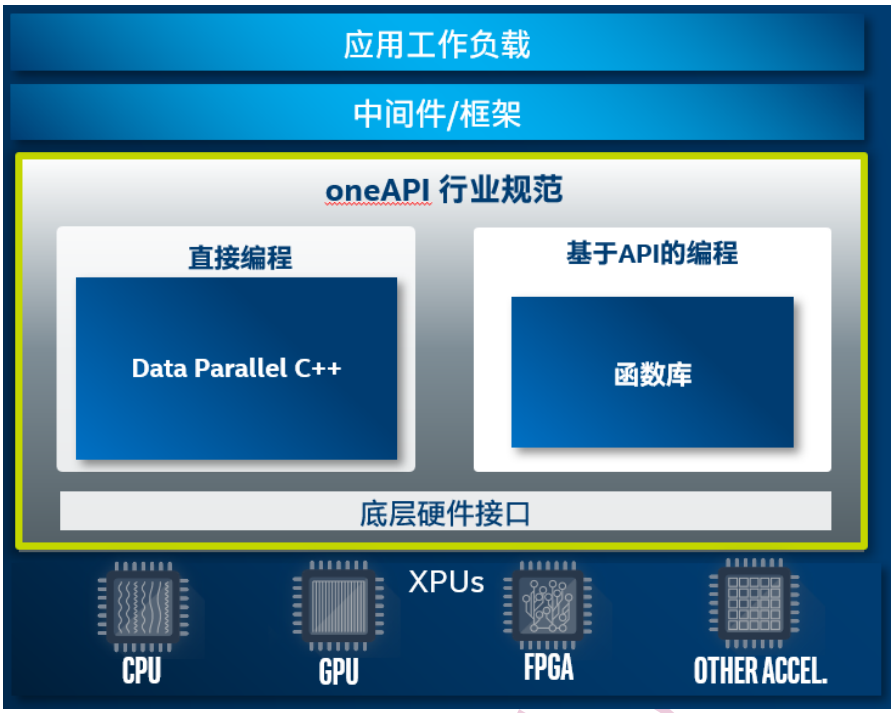
(一) oneAPI 及 Devcloud 简介

Intel oneAPI 是一个跨行业、开放、基于标准的统一的编程模型，旨在提供一个适用于各类计算架构的统一编程模型和应用程序接口。应用程序的开发者只需要开发一次代码，就可以让代码在跨平台的异构系统上执行，底层的硬件架构可以是 CPU、GPU、FPGA、神经网络处理器等。由此可见，使用 oneAPI 编写的程序既可以利用加速器提高程序性能，又具有可移植性。

oneAPI 为一系列的数据并行加速器提供了一个通用的开发者接口, 如下图:

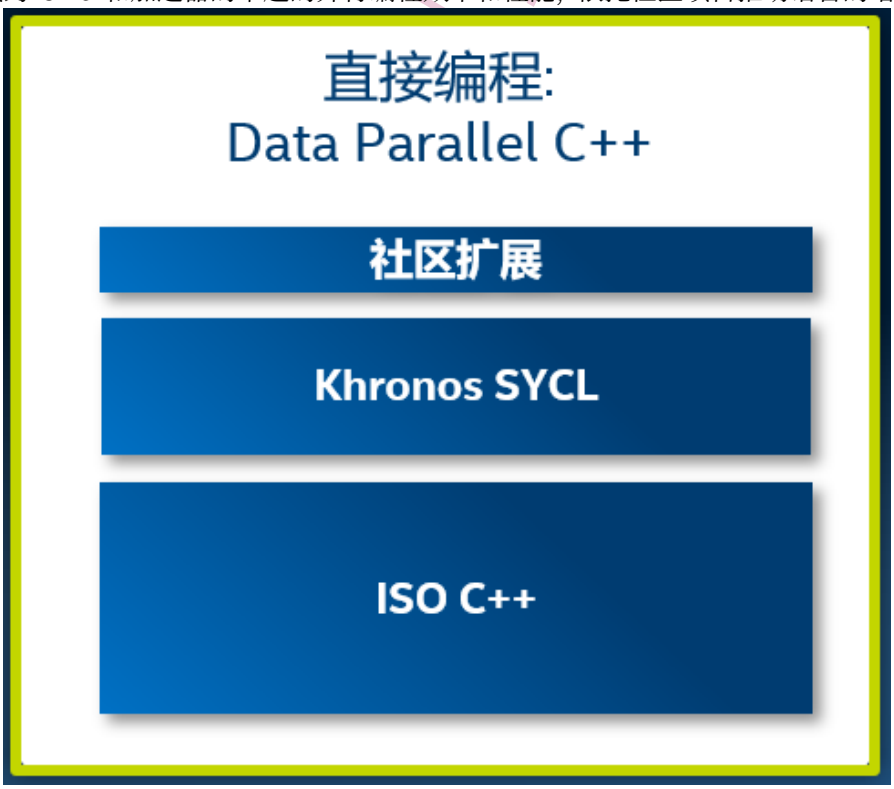


针对并行化表达对语言和函数库进行统一和简化，同时又不牺牲性能。基于行业标准和开放式规范，能够与现有 HPC 编程模型互操作。



英特尔 DevCloud: 一个可以在线开发 oneAPI 程序的平台, DevCloud 除了预装了 oneAPI 开发套件之外，还提供了有关 oneAPI 的教程，并且免费提供 GPU、FPGA 等加速器资源供我们使用，因此可以很方便地在 DevCloud 上学习 oneAPI 知识，并测试自己开发的 oneAPI 程序。

一个 oneAPI 运行环境由一个主机和一系列设备组成。主机通常是一个多核 CPU，而设备是一个或多个 GPU、FPGA，或是其他加速器。主机的处理器也可以进行并行计算。其主要组件是跨架构并行编程语言 DPC++（Data parallel C++）。它基于 ISO C++ 和 Khronos SYCL，提供跨 CPU 和加速器的卓越的并行编程效率和性能，依托社区项目推动语言的增强、不断演进。



形成了完成的开发生态，包括核心工具套件和库——Data Parallel C++ 编译器、库和分析工具，支持跨架构（CPU、GPU、FPGA）开发高性能应用：

Python 分发版包括 oneAPI 加速的 scikit-learn、NumPy 和 SciPy 库。

优化的性能库，支持线程化、数学、数据分析、深度学习和视频/图像/信号处理：

视频处理库：加速媒体、视频处理，跨架构的简单但强大的 API。

深度神经网络库：跨架构、跨操作系统支持，开源。

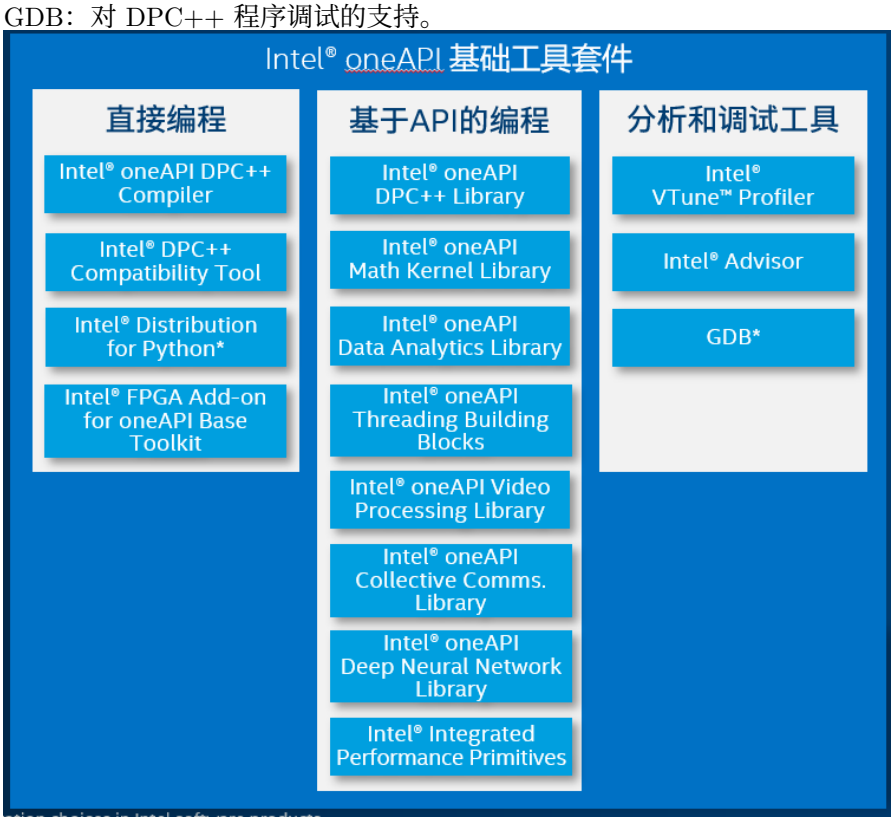
组通信库：利用组通信优化分布式机器学习，优化组通信实现。

分析调试工具：

Vtune Profiler：分析 DPC++ 程序，针对不同架构进行调优，OpenMP 卸载调优，多方面的性能剖析（线程、内存、cache、…），多种高级语言支持。

Advisor：估计卸载（到 GPU 等加速器）性能，优化访存和计算性能，向量化建议，多线程建议，流图分析。

GDB：对 DPC++ 程序调试的支持。



(二) oneAPI 特殊高斯消元

基于 Grober 基的特殊高斯消元问题描述

消元子、消元行，提高精度、效率算法分析

怎么利用 oneAPI 将串行算法改为并行算法

算法设计

并行化伪代码并行化代码

实验设计

串行与利用 oneAPI 并行化实现的特殊高斯消元性能对比

（三） 特殊高斯消元不同实现方式对比

SIMD 实现特殊高斯消元
多核 CPU 实现特殊高斯消元
性能对比及提升策略分析

（四） oneAPI 实现机制初探

框架
设计架构
实现方式