x40 并行编程指南

cucumber

2020-09-16

献给亲爱的你

目录

表格

插图

前言

随着核物理实验规模逐渐增大，数据量也飞速增长，单线程分析数据就比较慢了。

编写这份指南的目的是简单介绍并行编程技术，帮助大家愉快地处理数据。

这份指南将假定读者有 c 语言编程基础。本指南附带样例代码，并保证在指南发布日期代码是可编译、可执行的。

由于作者水平有限，代码风格鬼畜，可能出现各种错误/Undefined Behavior/代码风格杂糅/c 与 c++ 语法混杂等问题，望诸君海涵。

由于本指南并非用于出版，因此参考文献肯定是懒得标注的。

指南最开始随便介绍一下 x40 服务器现状。

指南第一部分介绍 c 语言的并行编程方法。其中 SIMD 的部分应该还是比较有用的，合理使用的话可以有不小的提速。多线程部分其实我也是现查的（因为我完全没使用过 c/c++ 的线程）。（顺带一提，我个人使用 golang 进行并行编程，其原生提供的协程和管道简单好用，有兴趣可以试一下。）

第二部分介绍使用 CUDA 的 GPGPU 编程。如果只是想利用 GPU 来进行并行加速，可以直接跳到这一部分。这里有 CUDA 的基本使用方法，还介绍了如何使用纹理来加速插值，还有如何使用流来隐藏数据传输延迟。

附录里面简单介绍了汇编语言，浮点数的计算机表示，还有高速缓存相关内容。未来可能加入更多内容。其实这部分也挺有趣的。在高速缓存的部分还有一个样例程序，可以体验一下胡乱访问数据造成的缓存不命中带来的性能下降。

在购买显卡之前，我曾在群里跟大家沟通过，有师兄提到现在大家分析数据还在使用单线程程序，多线程都没搞定，就要买显卡？听了以后我是痛心疾首。再观察一下，

大家的加速方法基本是同时运行好几个程序，这种比较原始的并行方法可能会占据过多的内存或带宽资源。

虽然我不觉得阅读了这个手册就能立即开始多线程分析，但是我希望这个可以帮助大家用“多线程”的方式思考如何解决问题，并在可以今后的程序中尝试使用并行。

本指南由 R 包 **knitr** (Xie, 2020b) 和 **bookdown** (Xie, 2020a) 生成。

本指南的许多知识源于《深入理解计算机系统》(Randal E. Bryant, 2010) 和 CUDA Toolkit Documentation (nVidia, 2020) .

《深入理解计算机系统》这本书还挺厉害的，是我以前旁听一门信科课程用的教材。虽然学得吊儿郎当，但是学完以后感觉对计算机系统的理解更深入了一点呢

关于 CUDA，有不懂就要取查手册，有问题就网上搜索一下。CUDA 的文档有些内容有点语焉不详，但是总体来说还可以，关于函数如何使用之类的，都可以查一下。

不知不觉就写了 100 多页，都快成一本小书了诶！

致谢

非常感谢老板们的支持，特别感谢杨老板提供资金为 x40 服务器购置显卡。

在安装显卡前，韩家兴花了许多时间备份了服务器硬盘，非常感谢他的付出。

在安装和备份之时服务器经常停止服务，也感谢同学们的理解。

最后，非常感谢各位读者花时间阅读本指南。希望这份指南能帮助您提升程序性能。

cucumber
于泡菜坛子中

作者简介

x40 服务器用户一名。科研偷懒中写一份指南。

本人在本科时关注到了 GPGPU 的大发展，心向往之。当时手里的笔记本只有 AMD 显卡，就尝试了一下 OpenCL. 但是后来也没有需要并行的程序，就再也没碰过了。

后来又解除了 CUDA，感觉用起来很简单，但是依旧没有什么实际用处。

到了 2018 年，我在日本。找老板配电脑的时候选了一个带 nvidia 显卡的型号。本来想的是学一个深度学习用来分析信号波形，然而最后也没学成，也没分析波形。但是我在其他方面发现了 GPU 的好处——聚类。做 PID 相当于是做聚类嘛。我找了个高斯混合模型，感觉用于 PID 非常合适，就写了 CUDA 程序，效果还不错。

如今，在计算粒子在磁场中的飞行轨迹，我又想到了 CUDA，实测效果很好。

~~好菜啊，没戏啦，科研啥的不干啦。~~

x40 服务器

本文将称我们 IP 地址结尾 64 服务器为 x40 服务器。

$64_{10} = 40_{16}$, 且其型号为 R940xa，称之为 x40 服务器简直合情合理！

0.1 基本硬件信息

CPU: 4 × Intel Xeon Gold 6136 @ 3.00GHz
GPU: 1 × Nvidia Titan V

CPU 共 48 核心 96 线程。GPU 有 5120 个 cuda 核心，12GB 显示内存。

第一部分

c 语言并行编程

这一部分借 c 语言并行编程之题，行介绍高级矢量扩展和线程之实。

这一部分的程序如无特殊说明，都能在 gcc7.3.0 下编译通过（如果我没有抄错的话）。

第一章　SIMD 和 SSE/AVX

SIMD (Single Instruction, Multiple Data)，即单指令多数据，顾名思义，是通过一条指令对多条数据进行同时操作。据维基百科说，最早得到广泛应用的 SIMD 指令集是 Intel 的 MMX 指令集，共包含 57 条指令。MMX 提供了 8 个 64 位的寄存器 (MM0 - MM7)，每个寄存器可以存放两个 32 位整数或 4 个 16 位整数或 8 个 8 位整数，寄存器中 “打包” 的多个数据可以通过一条指令同时处理，不再需要分成几次分别处理。

例如做 8 个 8 位整数加法 (c[i] = a[i] + b[i]. i {0,1,...,7})。

标量算法流程：
i = 0
将 a[i] 读入寄存器 0
将 b[i] 读入寄存器 1
对寄存器 0 和 1 内的 8 位整数求和并将结果存储在寄存器 0 中
将寄存器 0 中的八位整数写入 c[i] 的内存位置
i = i+1
比较 i 和 8
如果小于则重复以上步骤

矢量算法流程：
将 a[0-7] 读入矢量寄存器 0
将 b[0-7] 读入矢量寄存器 1
对矢量寄存器 0 和 1 按照 8 位一个整数同时求和，结果存储在
矢量寄存器 0 中
将适量寄存器 0 中的 64 位数据写入 c 的内存位置

如上例，对于 8 位整数的加法，理论上最大可以有 8 倍的提速。

之后，SSE 出现了，提供了 8 个 128 位寄存器 (XMM0 - XMM7)，并且有了处理浮点数的能力。可以同时处理两个双精度浮点数或四个单精度浮点数，或者同时处理四个 32 位整数或者八个 16 位整数又或者十六个 8 位整数。

再后来，又升级了 AVX。AVX 将 SSE 的每个 128 位寄存器扩展到 256 位，并增加了 8 个 256 寄存器。16 个 256 位寄存器称作 (YMM0 - YMM15)。再后来 Intel 又推出了 AVX512，把 YMM 扩展到 512 位，又新增 16 个寄存器，共 32 个 512 位寄存器 (ZMM0 - ZMM31)。(前几天 Linus 还怒斥了 Intel 的 AVX512)

x40 服务器是支持 AVX512 的，但是本指南不介绍 AVX512 使用方法 (主要是因为我也没用过)。但是原则上与 AVX 大同小异。而且，大概率您正在使用的桌面电脑或笔记本电脑也支持 AVX 指令集，但不大可能支持 AVX512，因此本章的代码您可以在自己的电脑上运行/测试。

1.1 一个简单的程序

単純な馬鹿にありたい。——『日常』

为便于演示 SIMD 并比较速度，我们创建一个非常非常简单的程序：拿三个 double 数组，第一组乘上第二组再加上第三组，结果存储在第四个数组中。想必各位读者都能很快实现这样的算法。下面列出本指南使用的程序（后续章节会基于这个程序进行改造）。为了加大处理压力，使运行时间可以观测到，这里强行让程序算 1,000,000 遍。可以在输出中看到计算耗时在 10s 左右。在输出运行时间后，挑出一些结果来对照一下看看对不对（嗯，在这个程序里上面和下面一毛一样，怎么会不对呢！）

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

__attribute__ ((noinline))
void muladd(double* a, double* b, double* c,
            double* d, unsigned long long N){
    unsigned long long i;
    for(i = 0; i < N; i++){
        d[i] = a[i] * b[i] + c[i];
```

```
    }
}

int main(){
    double* a;
    double* b;
    double* c;
    double* d;

    a = (double*)(malloc(8192*sizeof(double)));
    b = (double*)(malloc(8192*sizeof(double)));
    c = (double*)(malloc(8192*sizeof(double)));
    d = (double*)(malloc(8192*sizeof(double)));

    //Prepare data
    unsigned long long i;
    for(i = 0; i < 8192; i++){
        a[i] = (double)(rand()%2000) / 200.0;
        b[i] = (double)(rand()%2000) / 200.0;
        c[i] = ((double)i)/10000.0;
    }

    clock_t start, stop;
    double elapsed;
    start = clock();

    for(i = 0; i < 1000000; i++){
        muladd(a, b, c, d, 8192);
    }

    stop = clock();
    elapsed = (double)(stop-start) / CLOCKS_PER_SEC;
    printf("Elapsed time = %8.6f s\n", elapsed);
```

```
    for(i = 0; i < 8192; i++){
        if(i % 1001 == 0){
            printf("%5llu: %12.8f * %12.8f + %12.8f = %12.8f (%d)\n",
                   i, a[i], b[i], c[i], d[i], d[i]==a[i]*b[i]+c[i]);
        }
    }

    free(a);
    free(b);
    free(c);
    free(d);
}
```

> 关于程序随便提一句。`__attribute__ ((noinline))` 的作用是告诉编译器不要对这个函数 inline 展开。因为我们的 muladd 函数比较简单，可能会被编译器优化以后直接在调用处原地展开。虽然这样也没什么不行，但是为了后续分析程序行为，这里加上了这个标志。

编译运行一下，可以看到运算过程花了 20s 左右。

1.2 我的第一个 SIMD 程序！

> *May the 4s be with you.*

我们来使用 AVX，一次算四个。要显式地使用 AVX 指令集，可以使用所谓 “intrinsic”。这些 intrinsic 与 CPU 指令有直接的对应。可以参考 Intel Intrinsics Guide[1]。要使用这些 intrinsic，需要 `#include <x86intrin.h>`. 在编译的时候，还要加上`-mavx`

```
gcc -mavx -o muladd_simd muladd_simd.c
```

改造后的程序是这个样子：

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <x86intrin.h>

__attribute__ ((noinline))
void muladd(double* a, double* b, double* c,
            double* d, unsigned long long N){
    unsigned long long i, j;
    unsigned long long M = N>>2;
    for(i = 0; i < M; i++){
        __m256d ymma = _mm256_load_pd(a+i*4);
        __m256d ymmb = _mm256_load_pd(b+i*4);
        __m256d ymmc = _mm256_load_pd(c+i*4);
        __m256d ymmd = _mm256_mul_pd(ymma, ymmb);
        ymmd = _mm256_add_pd(ymmd, ymmc);
        _mm256_store_pd(d+i*4,ymmd);
    }
    for(i = N-N%4; i < N; i++){
        d[i] = a[i] * b[i] + c[i];
    }
}

int main(){
    double* a;
    double* b;
    double* c;
    double* d;

    a = (double*)(aligned_alloc(32,8192*sizeof(double)));
    b = (double*)(aligned_alloc(32,8192*sizeof(double)));
    c = (double*)(aligned_alloc(32,8192*sizeof(double)));
    d = (double*)(aligned_alloc(32,8192*sizeof(double)));
```

```
    //Prepare data
    unsigned long long i;
    for(i = 0; i < 8192; i++){
        a[i] = (double)(rand()%2000) / 200.0;
        b[i] = (double)(rand()%2000) / 200.0;
        c[i] = ((double)i)/10000.0;
    }

    clock_t start, stop;
    double elapsed;
    start = clock();

    for(i = 0; i < 1000000; i++){
        muladd(a, b, c, d, 8192);
    }

    stop = clock();
    elapsed = (double)(stop-start) / CLOCKS_PER_SEC;
    printf("Elapsed time = %8.6f s\n", elapsed);
    for(i = 0; i < 8192; i++){
        if(i % 1001 == 0){
            printf("%5llu: %16.8f * %16.8f + %16.8f = %16.8f (%d)\n",
                   i, a[i], b[i], c[i], d[i], d[i]==a[i]*b[i]+c[i]);
        }
    }

    free(a);
    free(b);
    free(c);
    free(d);
}
```

关于这段程序又要提一句。`malloc(8192*sizeof(double))` 换成了

`aligned_alloc(32, 8192*sizeof(double))`. 这是因为 `_mm256_load_pd` 要求内存是对齐到 256bit 的 (YMM 寄存器的宽度是 256bit).`aligned_alloc` 函数可以实现开辟对齐到指定地址的内存空间，第一个参数是对齐方式，单位是字节 (Byte)(32Byte × 8bit/byte = 256 bit)。第二个参数是内存大小。要注意第二个参数必须是第一个的整数倍。
另外,也可以直接 `malloc`,而把 `_mm256_load_pd` 替换为 `_mm256_load_upd`. u 代表 “unaligned”。

以下太长不看
读取双精度浮点数据到 YMM 有两个指令，VMOVAPD 和 VMOVUPD。VMOVAPD 要求内存地址是对齐到 256 位的，而 VMOVUPD 无此要求。据说在早期的支持 AVX 的 cpu 上 VMOVAPD 要快一些，似乎在新一些的平台上性能已经相当了。（但我没有详细考证过。）以及，`_mm256_load_upd` 似乎并不会被翻译为 vmovupd，而是变成了两条指令，大致是先读入两个 double，再把另外两个插入进来。emm...

赶紧编译运行一下，这回可快了，只要……嗯？要 16s？4 个 double 同时算，这提升也太小了吧？咱们看一下编译出来的内容：

```
$ gdb muladd_simd
  (gdb) disassemble muladd
  Dump of assembler code for function muladd:
   0x0000000000400657 <+0>:     lea    0x8(%rsp),%r10
   0x000000000040065c <+5>:     and    $0xfffffffffffffe0,%rsp
   0x0000000000400660 <+9>:     pushq  -0x8(%r10)
   0x0000000000400664 <+13>:    push   %rbp
   0x0000000000400665 <+14>:    mov    %rsp,%rbp
   0x0000000000400668 <+17>:    push   %r10
   0x000000000040066a <+19>:    sub    $0x150,%rsp
   0x0000000000400671 <+26>:    mov    %rdi,-0x198(%rbp)
   0x0000000000400678 <+33>:    mov    %rsi,-0x1a0(%rbp)
   0x000000000040067f <+40>:    mov    %rdx,-0x1a8(%rbp)
   0x0000000000400686 <+47>:    mov    %rcx,-0x1b0(%rbp)
   0x000000000040068d <+54>:    mov    %r8,-0x1b8(%rbp)
```

```
0x0000000000400694 <+61>:    mov    -0x1b8(%rbp),%rax
0x000000000040069b <+68>:    shr    $0x2,%rax
0x000000000040069f <+72>:    mov    %rax,-0x20(%rbp)
0x00000000004006a3 <+76>:    movq   $0x0,-0x18(%rbp)
0x00000000004006ab <+84>:    jmpq   0x4007e5 <muladd+398>
0x00000000004006b0 <+89>:    mov    -0x18(%rbp),%rax
0x00000000004006b4 <+93>:    shl    $0x5,%rax
0x00000000004006b8 <+97>:    mov    %rax,%rdx
0x00000000004006bb <+100>:   mov    -0x198(%rbp),%rax
0x00000000004006c2 <+107>:   add    %rdx,%rax
0x00000000004006c5 <+110>:   mov    %rax,-0x188(%rbp)
0x00000000004006cc <+117>:   mov    -0x188(%rbp),%rax
0x00000000004006d3 <+124>:   vmovapd (%rax),%ymm0
0x00000000004006d7 <+128>:   vmovapd %ymm0,-0x50(%rbp)
0x00000000004006dc <+133>:   mov    -0x18(%rbp),%rax
0x00000000004006e0 <+137>:   shl    $0x5,%rax
0x00000000004006e4 <+141>:   mov    %rax,%rdx
0x00000000004006e7 <+144>:   mov    -0x1a0(%rbp),%rax
0x00000000004006ee <+151>:   add    %rdx,%rax
0x00000000004006f1 <+154>:   mov    %rax,-0x180(%rbp)
0x00000000004006f8 <+161>:   mov    -0x180(%rbp),%rax
0x00000000004006ff <+168>:   vmovapd (%rax),%ymm0
0x0000000000400703 <+172>:   vmovapd %ymm0,-0x70(%rbp)
0x0000000000400708 <+177>:   mov    -0x18(%rbp),%rax
0x000000000040070c <+181>:   shl    $0x5,%rax
0x0000000000400710 <+185>:   mov    %rax,%rdx
0x0000000000400713 <+188>:   mov    -0x1a8(%rbp),%rax
0x000000000040071a <+195>:   add    %rdx,%rax
0x000000000040071d <+198>:   mov    %rax,-0x178(%rbp)
0x0000000000400724 <+205>:   mov    -0x178(%rbp),%rax
0x000000000040072b <+212>:   vmovapd (%rax),%ymm0
0x000000000040072f <+216>:   vmovapd %ymm0,-0x90(%rbp)
0x0000000000400737 <+224>:   vmovapd -0x50(%rbp),%ymm0
```

```
0x000000000040073c <+229>:   vmovapd %ymm0,-0x150(%rbp)
0x0000000000400744 <+237>:   vmovapd -0x70(%rbp),%ymm0
0x0000000000400749 <+242>:   vmovapd %ymm0,-0x170(%rbp)
0x0000000000400751 <+250>:   vmovapd -0x150(%rbp),%ymm0
0x0000000000400759 <+258>:   vmulpd -0x170(%rbp),%ymm0,%ymm0
0x0000000000400761 <+266>:   vmovapd %ymm0,-0xb0(%rbp)
0x0000000000400769 <+274>:   vmovapd -0xb0(%rbp),%ymm0
0x0000000000400771 <+282>:   vmovapd %ymm0,-0x110(%rbp)
0x0000000000400779 <+290>:   vmovapd -0x90(%rbp),%ymm0
0x0000000000400781 <+298>:   vmovapd %ymm0,-0x130(%rbp)
0x0000000000400789 <+306>:   vmovapd -0x110(%rbp),%ymm0
0x0000000000400791 <+314>:   vaddpd -0x130(%rbp),%ymm0,%ymm0
0x0000000000400799 <+322>:   vmovapd %ymm0,-0xb0(%rbp)
0x00000000004007a1 <+330>:   mov    -0x18(%rbp),%rax
0x00000000004007a5 <+334>:   shl    $0x5,%rax
0x00000000004007a9 <+338>:   mov    %rax,%rdx
0x00000000004007ac <+341>:   mov    -0x1b0(%rbp),%rax
0x00000000004007b3 <+348>:   add    %rdx,%rax
0x00000000004007b6 <+351>:   mov    %rax,-0xb8(%rbp)
0x00000000004007bd <+358>:   vmovapd -0xb0(%rbp),%ymm0
0x00000000004007c5 <+366>:   vmovapd %ymm0,-0xf0(%rbp)
0x00000000004007cd <+374>:   mov    -0xb8(%rbp),%rax
0x00000000004007d4 <+381>:   vmovapd -0xf0(%rbp),%ymm0
0x00000000004007dc <+389>:   vmovapd %ymm0,(%rax)
0x00000000004007e0 <+393>:   addq   $0x1,-0x18(%rbp)
0x00000000004007e5 <+398>:   mov    -0x18(%rbp),%rax
0x00000000004007e9 <+402>:   cmp    -0x20(%rbp),%rax
0x00000000004007ed <+406>:   jb     0x4006b0 <muladd+89>
0x00000000004007f3 <+412>:   mov    -0x1b8(%rbp),%rax
0x00000000004007fa <+419>:   and    $0xfffffffffffffffc,%rax
0x00000000004007fe <+423>:   mov    %rax,-0x18(%rbp)
0x0000000000400802 <+427>:   jmp    0x400879 <muladd+546>
0x0000000000400804 <+429>:   mov    -0x18(%rbp),%rax
```

```
   0x0000000000400808 <+433>:   lea    0x0(,%rax,8),%rdx
   0x0000000000400810 <+441>:   mov    -0x198(%rbp),%rax
   0x0000000000400817 <+448>:   add    %rdx,%rax
   0x000000000040081a <+451>:   vmovsd (%rax),%xmm1
   0x000000000040081e <+455>:   mov    -0x18(%rbp),%rax
   0x0000000000400822 <+459>:   lea    0x0(,%rax,8),%rdx
   0x000000000040082a <+467>:   mov    -0x1a0(%rbp),%rax
   0x0000000000400831 <+474>:   add    %rdx,%rax
   0x0000000000400834 <+477>:   vmovsd (%rax),%xmm0
   0x0000000000400838 <+481>:   vmulsd %xmm0,%xmm1,%xmm0
   0x000000000040083c <+485>:   mov    -0x18(%rbp),%rax
   0x0000000000400840 <+489>:   lea    0x0(,%rax,8),%rdx
   0x0000000000400848 <+497>:   mov    -0x1a8(%rbp),%rax
   0x000000000040084f <+504>:   add    %rdx,%rax
   0x0000000000400852 <+507>:   vmovsd (%rax),%xmm1
   0x0000000000400856 <+511>:   mov    -0x18(%rbp),%rax
   0x000000000040085a <+515>:   lea    0x0(,%rax,8),%rdx
   0x0000000000400862 <+523>:   mov    -0x1b0(%rbp),%rax
   0x0000000000400869 <+530>:   add    %rdx,%rax
   0x000000000040086c <+533>:   vaddsd %xmm1,%xmm0,%xmm0
   0x0000000000400870 <+537>:   vmovsd %xmm0,(%rax)
   0x0000000000400874 <+541>:   addq   $0x1,-0x18(%rbp)
   0x0000000000400879 <+546>:   mov    -0x18(%rbp),%rax
   0x000000000040087d <+550>:   cmp    -0x1b8(%rbp),%rax
   0x0000000000400884 <+557>:   jb     0x400804 <muladd+429>
   0x000000000040088a <+563>:   nop
   0x000000000040088b <+564>:   add    $0x150,%rsp
   0x0000000000400892 <+571>:   pop    %r10
   0x0000000000400894 <+573>:   pop    %rbp
   0x0000000000400895 <+574>:   lea    -0x8(%r10),%rsp
   0x0000000000400899 <+578>:   retq
End of assembler dump.
```

对比一下标量版得到的汇编：

```
$ gdb muladd
  (gdb) disassemble muladd
  Dump of assembler code for function muladd:
   0x0000000000400637 <+0>:     push   %rbp
   0x0000000000400638 <+1>:     mov    %rsp,%rbp
   0x000000000040063b <+4>:     mov    %rdi,-0x18(%rbp)
   0x000000000040063f <+8>:     mov    %rsi,-0x20(%rbp)
   0x0000000000400643 <+12>:    mov    %rdx,-0x28(%rbp)
   0x0000000000400647 <+16>:    mov    %rcx,-0x30(%rbp)
   0x000000000040064b <+20>:    mov    %r8,-0x38(%rbp)
   0x000000000040064f <+24>:    movq   $0x0,-0x8(%rbp)
   0x0000000000400657 <+32>:    jmp    0x4006c2 <muladd+139>
   0x0000000000400659 <+34>:    mov    -0x8(%rbp),%rax
   0x000000000040065d <+38>:    lea    0x0(,%rax,8),%rdx
   0x0000000000400665 <+46>:    mov    -0x18(%rbp),%rax
   0x0000000000400669 <+50>:    add    %rdx,%rax
   0x000000000040066c <+53>:    movsd  (%rax),%xmm1
   0x0000000000400670 <+57>:    mov    -0x8(%rbp),%rax
   0x0000000000400674 <+61>:    lea    0x0(,%rax,8),%rdx
   0x000000000040067c <+69>:    mov    -0x20(%rbp),%rax
   0x0000000000400680 <+73>:    add    %rdx,%rax
   0x0000000000400683 <+76>:    movsd  (%rax),%xmm0
   0x0000000000400687 <+80>:    mulsd  %xmm1,%xmm0
   0x000000000040068b <+84>:    mov    -0x8(%rbp),%rax
   0x000000000040068f <+88>:    lea    0x0(,%rax,8),%rdx
   0x0000000000400697 <+96>:    mov    -0x28(%rbp),%rax
   0x000000000040069b <+100>:   add    %rdx,%rax
   0x000000000040069e <+103>:   movsd  (%rax),%xmm1
   0x00000000004006a2 <+107>:   mov    -0x8(%rbp),%rax
   0x00000000004006a6 <+111>:   lea    0x0(,%rax,8),%rdx
   0x00000000004006ae <+119>:   mov    -0x30(%rbp),%rax
   0x00000000004006b2 <+123>:   add    %rdx,%rax
   0x00000000004006b5 <+126>:   addsd  %xmm1,%xmm0
```

```
   0x00000000004006b9 <+130>:   movsd  %xmm0,(%rax)
   0x00000000004006bd <+134>:   addq   $0x1,-0x8(%rbp)
   0x00000000004006c2 <+139>:   mov    -0x8(%rbp),%rax
   0x00000000004006c6 <+143>:   cmp    -0x38(%rbp),%rax
   0x00000000004006ca <+147>:   jb     0x400659 <muladd+34>
   0x00000000004006cc <+149>:   nop
   0x00000000004006cd <+150>:   pop    %rbp
   0x00000000004006ce <+151>:   retq
End of assembler dump.
```

差好多！仔细看一下，SIMD 版本中 <+89> 到 <+128> 这部分。简单解释一下：

	汇编	含义
< +89>	mov -0x18(%rbp),%rax	将循环变量 (i) 读入寄存器 rax
< +93>	shl $0x5,%rax	rax 中的数左移 5 位 (相当于 *32).
< +97>	mov %rax,%rdx	将 rax 中的数拷贝到 rdx 寄存器中
<+100>	mov -0x198(%rbp),%rax	将数组 A 的地址读入寄存器 rax
<+107>	add %rdx,%rax	rax = rax + rdx
<+124>	vmovapd (%rax),%ymm0	从 rax 寄存器所示地址处读取 256 位到 YMM0
<+128>	vmovapd %ymm0,-0x50(%rbp)	将 YMM0 中的数据保存在栈上的一个位置

> 如果您曾经学习过《汇编语言程序设计》或者《微机原理》等课程，可能会对上述汇编代码有疑问。这是由于通常微机原理课程使用的是 Intel 格式的汇编 (ins dst, src)，而 GCC 使用 AT&T 格式 (ins src, dst)，一般来讲操作数的顺序和 Intel 格式都是反过来的。
> 如果您并不懂汇编，也可以直接看结论。如果对汇编感兴趣，可以参考附录。

简单来说，就是每一次循环把需要的值取到寄存器中，再存到栈上 (可以理解为函数自有的内存)。三个数据都存好后，再把他们依次从栈里取到不同的 YMM 寄存器里，做求积和求和，再存到相应位置。（这只是大致过程，实际看一看，它干的傻事还不少）。

仔细想想，其实编译器这么做是有道理的。在程序里，我们写了 `__m256d ymma = _mm256_load_pd(a+i*4);`，这意味着我们要有一个 `ymma` 变量。编译器并不知道我

们会不会对这个变量做·一·些·奇·怪·的·事·因此便把它又写进了栈里。那么有没有办法让编译器明白我们不会乱动 ymma 呢？有！

1.3 搞个寄存器变量

> 我买几个橘子去。你就在此地，不要走动。——《背影》

只要在声明变量的时候，在类型前面加上 register 关键字，编译器就明白这个东西应该常驻在寄存器里面，就不会来回读写内存了。需要注意**不能对寄存器变量取地址！**顺便，这个浓眉大眼的循环次数也可以放到寄存器里面的样子，把这些东西加上 register 关键字：

```
register unsigned long long M = N>>2;
for(i = 0; i < M; i++){
    register __m256d ymma = _mm256_load_pd(a+i*4);
    register __m256d ymmb = _mm256_load_pd(b+i*4);
    register __m256d ymmc = _mm256_load_pd(c+i*4);
    register __m256d ymmd = _mm256_mul_pd(ymma, ymmb);
    ymmd = _mm256_add_pd(ymmd, ymmc);
    _mm256_store_pd(d+i*4,ymmd);
}
```

再来运行下。**别忘了编译时加-mavx**
大概 12 秒左右。已经比最原始的版本快了近一倍了！四舍五入一个亿啊！

> 在新标准的 c++ 中，据说 register 关键字已经没有意义了，加了 register 编译器也不一定用寄存器，不加也不一定就不用。但是在我用 c++ 测试的时候，发现还是有用的。
> 在 g++ 里增加-std=c++11 或者-std=c++14 都可以顺利编译，运行速度一致。如果加-std=c++17 则会报几个 warning，说是新标准不许 regester，但是还是会比不加 register 关键字要快。（啊哈，你个 c++17，嘴上说着 warning，身体倒是很老实嘛！）

1.4 我比编译器聪明系列

コンピューターも、天国へいけるかな。——『*Lost Universe*』

还能不能再快一点？我想要的四倍提速呢？
下面我们使用 c 语言的内联汇编做这件事。

```
__attribute__ ((noinline))
void muladd(double* a, double* b, double* c,
            double* d, unsigned long long N){
    unsigned long long i;
    __asm__ __volatile__(
            "movq %0, %%rax \n\t"
            "movq %1, %%rbx \n\t"
            "movq %2, %%rcx \n\t"
            "movq %3, %%rdx \n\t"
            "movq %4, %%r8  \n\t"
            "shr  $2, %%r8  \n\t"
            "movq $0, %%r9  \n\t"
            "jmp  .check_%= \n\t"
            ".loop_%=:          \n\t"
            "shl $2, %%r9   \n\t"
            "vmovupd (%%rax, %%r9, 8), %%ymm0 \n\t"
            "vmovupd (%%rbx, %%r9, 8), %%ymm1 \n\t"
            "vmovupd (%%rcx, %%r9, 8), %%ymm2 \n\t"
            "vmulpd %%ymm0, %%ymm1, %%ymm3    \n\t"
            "vaddpd %%ymm2, %%ymm3, %%ymm3    \n\t"
            "vmovupd %%ymm3, (%%rdx, %%r9, 8) \n\t"
            "shr $2, %%r9                     \n\t"
            "add $1, %%r9                     \n\t"
            ".check_%=:                       \n\t"
            "cmpq %%r8, %%r9                  \n\t"
            "jl .loop_%=                      \n\t"
            :
```

```
            :"m"(a), "m"(b), "m"(c), "m"(d), "m"(N)
            :"%rax", "%rbx", "%rcx", "%rdx", "%r8", "%r9",
             "%ymm0", "%ymm1", "%ymm2", "%ymm3", "memory"
            );
    if(N%4!=0){
        for(i = N-N%4; i<N; i++){
            d[i] = a[i]*b[i]+c[i];
        }
    }
}
```

> 关于内联汇编的介绍，请参见附录。这里用一段汇编代替了前面的循环部分。当 N 不是 4 的整数倍时余下的 1-3 个运算使用普通 c 代码，并不会对性能产生太大影响。以及，这次编译不需要`-mavx` 了。

编译，运行一下。不到 4s！达到了预期的速度。

当然，也可以找一下 AVX512 指令怎么写（其实跟 AVX 是基本一样的），获得进一步加速。

1.5 编译器比我聪明系列

Doing nothing often leads to the very best something.——Winnie-the-Pooh

虽然这里手写了汇编，但是在一般的程序里不建议这么做，除非有信心做得好并且能加速。

实测，`gcc -mavx -O3 -o muladd muladd.c` 编译出来的程序与上述内联汇编的程序速度相差无几。在`-O3` 优化下 gcc 是有矢量化优化的，也是把运算优化到了四个一组同时做。

同理，`gcc -mavx512f -O3 -o muladd muladd.c` 会进行 AVX512 矢量化，八个一组，再快一倍（可能不到点一倍）。

此外，使用 intrinsic 的同时，如果打开`-O2` 优化，速度也会很快，甚至比手写汇编快那么一点。看一下反汇编，是少了一些取数到寄存器的操作（好吧我承认我学艺不

精）。但是-O2 优化本身是不进行矢量化的，也就是说在-O2 优化下如果使用我们最开始的 muladd.c 的话是利用不到 SIMD 来加速的。因此用-O2 配合 intrinsic 可能是一个好的选择。

> 顺便提一下，无论是汇编还是 intrinsic，可读性都比较差，最好多加一些注释。

1.6 本章小结

SIMD 大概是粒度最小的并行了吧？如同上面的例子，适当使用已经可以极大地提高程序运行速度了。

要注意一点，这里面“矢量运算”只是同时做好几个运算地意思，如果你想真的把它当作矢量，甚至想求个外积，那就超出处理器的能力范围了。只能把外积运算分解成加减乘除再进行计算。

这一章到这里就应该结束了，但我觉得阅读指南的你脑中一定充满了问号。所以在这里尝试自问自答一下。

问：为什么要做 1,000,000 次 8192 维数组的运算，而不直接用 8,192,000,000 维的数组？

答：是的。在我制作这份指南之前也是这么想的。但是发现 SIMD 以后速度并没有提高。经查，发现原理在于内存带宽吃满了。虽然处理器可以算得飞快，但是内存速度跟不上导致速度被限制了，因此矢量算法和标量算法速度相近。而减小数据规模以后，8192 个 double 都会存储在 CPU 的高速缓存中，这样就避免了内存带宽的限制，从而可以比较向量和标量指令的速度差异。
在实际使用中，对数据的处理显然不只是先求积再求和这种简单运算。随着运算变得复杂，当处理数据的时间达到甚至超过内存存取用时的时候，SIMD 的提速效果自然就可以体现出来了。

此外，$8,192,000,000 double \times 8 Byte/double = 65536 \times 10^6 Byte \approx 64 GB$ 吃内存稍微有点大。

问：既然 gcc 可以优化得这么好，搞这么复杂就没有意义了吧？

答：这可不好说了。在很久以前，听说 gcc 的-O3 有可能会搞出莫名其妙的 bug。虽然我没遇到过这种情况，虽然这种情况肯定非常少见，但是总的来说给人一种不太可靠的感觉。此外，纯凭个人感觉，可能 gcc 的优化力量也是有限的，如果计算过程比较长比较复杂，可能 gcc 并不一定能选出最合适的优化方法（可能会找不到数据/变量之间的关联?）。还有，如果你的程序不使用 C/C++ 的话，还有一定的可能你的编译器并不支持 AVX，这时就要自己动手丰衣足食（当然也可以用 C/C++ 搞一个动态链接库）。另外`-O2` 不会自动进行矢量化，所以至少 intrinsic 在`-O2` 下还是有意义的。不开优化开关的话，还是要汇编才快一些。具体怎么写代码用什么优化还是要自己取舍。

问：作者你好厉害哦！咋啥都懂？

答：正常小朋友一般问不出来这种问题。

1.7 补充内容

在这里补充一些比较常用 (大概) 的 AVX 指令和对应的 Intrinsic。（指令都是 AT&T 格式）

Intrinsic	指令	描述
_mm256_loadu_pd	VMOVUPD m256, ymm	将内存中连续的 4 个 double 读入 ymm
_mm256_storeu_pd	VMOVUPD ymm, m256	将 ymm 寄存器中的 4 个 double 写入连续内存
_mm256_add_pd	VADDPD ymm1, ymm2, ymm3	ymm3 = ymm2 + ymm1
_mm256_sub_pd	VSUBPD ymm1, ymm2, ymm3	ymm3 = ymm2 - ymm1
_mm256_mul_pd	VMULPD ymm1, ymm2, ymm3	ymm3 = ymm2 * ymm1
_mm256_div_pd	VDIVPD ymm1, ymm2, ymm3	ymm3 = ymm2 / ymm1

Intrinsic	指令	描述
_mm256_permute4x64_pd	VPERMPD imm8, ymm1, ymm2	ymm1 按照 imm8 所述重排入 ymm2

解释一下除法和减法反过来的问题。前面提到 AT&T 汇编和 Intel 汇编是反向的。在 MOV 之类的指令中，AT&T 格式看起来和谐一点，结果就是在减法和除法的地方反过来了。

关于 permute，imm8 是一个立即数，每两位为一组，表示 ymm 中的第某个数。

例：
ymm1 = (a, b, c, d)
VPEMPD $0xc5, ymm1, ymm2
=> ymm2 = (b, b, a, d)

是这样的，0xc5 = 11 00 01 01 (b). 00 对应 a，01 对应 b，10 对应 c，11 对应 d。而从 MSB 到 LSB 的顺序是 (3,2,1,0) 的顺序。因此目标的第 0 个 double 是根据最低两位 01 取数据，取到 b；第一个同理，01->b；第二个是 00，取 a；第三个是 11，取 d。另外，如果内存里连续的 double 是 (a,b,c,d) 的话，VMOVUPD 取进 ymm 寄存器的值就是 (a,b,c,d). 如果参考一些汇编手册，可能会发现手册里写的是反过来的 (d,c,b,a). 这只是表示方法不同。在汇编手册里习惯按照低位在右高位在左的顺序表示。

第二章　c 语言多线程编程

这部分我也是现学现卖。C11 标准引入了线程支持，但是直到 glibc 的 2.28 版本才实现了 C11 标准的线程。不幸服务器上的 glibc 版本是 2.17。

其实在 linux 下，c 语言早已有线程库 pthread 了。据说 glibc 里的线程库就是直接把 pthread 封装了一下
(～￣　￣)～

其实 pthread 和 thread 我都不会用，为了拥抱新标准简单学习了一下 C11 的 thread，这里就简单讲一下。

2.1　还是乘加运算

大切な人といつかまた巡り会えますように。——『*Plastic Memories*』

这里是 4 个线程进行运算的 c 代码

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <threads.h>

struct Input{
    double* a;
    double* b;
    double* c;
```

```
    double* d;
    unsigned long long N;
    unsigned long long R;
};

__attribute__ ((noinline))
int muladd_th(void* input){
    struct Input* in;
    unsigned long long i, j;
    double* va;
    double* vb;
    double* vc;
    double* vd;
    in = (struct Input*)input;
    va = in->a;
    vb = in->b;
    vc = in->c;
    vd = in->d;
    for(i = 0; i < in->R; i++){
        for(j = 0; j < in->N; j++){
            vd[j] = va[j] * vb[j] + vc[j];
        }
    }
    return 0;
}

__attribute__ ((noinline))
void muladd(double* a, double* b, double* c,
            double* d, unsigned long long N,
            unsigned long long R){
    thrd_t threads[4];
    struct Input inputs[4];
    unsigned long long i;
```

```
    for(i = 0; i < 4; i++){
        inputs[i].a = a + N/4*i;
        inputs[i].b = b + N/4*i;
        inputs[i].c = c + N/4*i;
        inputs[i].d = d + N/4*i;
        inputs[i].N = N/4;
        inputs[i].R = R;
        if(i == 3){
            inputs[i].N = N-N/4*3;
        }
        thrd_create(&(threads[i]), muladd_th, &(inputs[i]));
    }
    for(i = 0; i < 4; i++){
        thrd_join((threads[i]), NULL);
    }

}

int main(){
    double* a = (double*)(malloc(8192*sizeof(double)));
    double* b = (double*)(malloc(8192*sizeof(double)));
    double* c = (double*)(malloc(8192*sizeof(double)));
    double* d = (double*)(malloc(8192*sizeof(double)));

    //Prepare data
    unsigned long long i;
    for(i = 0; i < 8192; i++){
        a[i] = (double)(rand()%2000) / 200.0;
        b[i] = (double)(rand()%2000) / 200.0;
        c[i] = ((double)i)/10000.0;
    }

    struct timespec start, stop;
```

```
    double elapsed;
    clock_gettime(CLOCK_MONOTONIC, &start);

    muladd(a, b, c, d, 8192, 1000000);
    clock_gettime(CLOCK_MONOTONIC, &stop);
    elapsed = (double)(stop.tv_sec-start.tv_sec);
    elapsed += (double)(stop.tv_nsec-start.tv_nsec)/ 1000000000.0;
    printf("Elapsed time = %8.6f s\n", elapsed);
    for(i = 0; i < 8192; i++){
        if(i % 1001 == 0){
            printf("%5llu: %16.8f * %16.8f + %16.8f = %16.8f (%d)\n",
                   i, a[i], b[i], c[i], d[i], d[i]==a[i]*b[i]+c[i]);
        }
    }

    free(a);
    free(b);
    free(c);
    free(d);
}
```

简单说一下。需要 `#include <threads.h>`。这里把 4096 维数组拆成 4 份，分别由四个线程完成计算。1,000,000 次的循环也放到线程里面做。此外，这里还更换了计时函数。由于之前使用的计时函数在多线程的情况下会把每个线程的时间加起来，计算的时间就不对了。如果亲自试一试的话，会发现感受到的时间比输出的时间要短。

此外，上述程序需要 glibc>=2.28. 要再 x40 上运行的话可以自己编译一个 glibc。我尝试了编译 glibc 2.30。glibc 2.30 需要的 gmake 版本比 x40 带有的要高一些，所以又得自己编译一个 gmake。简单说一下流程：

下载最新的 gnu make 的源代码，再里面直接

```
$ ./configure
$ make
```

得到一个 make 的可执行文件。
将这个 make 重命名为 gmake 并拷贝到一个地方，比如 ~/bin/
再把这个地方加入到 $PATH 环境变量里 export PATH=~/bin/:$PATH.
然后下载 glibc 2.30 的源代码，解压。在其目录里面创建一个 build 文件夹并进入。

```
$ ../configure --prefix=~/glibc230/
$ make -j10 all
$ make install
```

注意一定要指定--prefix, 否则 make install 会尝试将 glibc 安装到系统目录。
这里假设你将 glibc 安装到 ~/glibc230.
安装好以后就可以编译 muladd_mt.c 了

```
gcc -L~/glibc230/lib -I~/glibc230/include \
-Wl,--rpath=~/glibc230/lib \
-Wl,--dynamic-linker=~/glibc230/lib/ld-linux-x86-64.so.2 \
-std=c11 \
-o muladd_mt muladd_mt.c -pthread
```

结果，用时 6 秒左右。

注意！后面的 thrd_join((threads[i]), NULL); 非常重要，一定不可省略。这句话保证了调用该函数的线程在这里等，直到 threads[i] 的线程结束。第二个参数如果不是 NULL 的话，threads[i] 线程会把返回值写到第二参数指示的地址。如果不写 thrd_join 会导致主线程提前退出，于是 threads[i] 线程的运行结果就拿不到了。这是不好的。

thrd_create 函数的三个参数：
第一个是一个指向 thrd_t 结构的指针，后面 thrd_join 等操作都需要这个 thrd_t 结构，相当于线程的编号。

第二个参数是一个函数指针 (直接用函数名即可)。这个函数必须是接受一个 `void*` 参数并返回一个 `int` 的函数。`void*` 参数用来向线程传递数据。
第三个参数就是要向上述函数传入的参数。由于参数需要是一个指针，因此在上面的程序里面设计了一个结构 `Input`，里面包含了计算所需的信息。
创建完后线程就运行起来了。这时该干啥干啥，然后用 `thrd_join` 等结果就行了。

2.2 本章小结

又是这个环节！线程比 SIMD 灵活多了，每个线程可以做不同的事情（可以根据传入的参数判断一下）。线程们在多核心 CPU 上可以同时执行，可以看到同样的内存空间。用个循环把它们安排明白就可以开始等了

相信你的脑中又一次充满了问号，我再尝试自问自答一下。

问：这里也是 muladd 函数，为什么不直接把它执行 1,000,000 遍？
答：这是因为线程的创建和销毁是有代价的。频繁创建和销毁线程会占用过多的资源。最好是创建好线程后让它执行一整套任务。此外，有一种“线程池”的技术，可以先创建一个包含一些线程的池，然后向线程池发送任务。接到任务后线程池会自动唤醒一个线程取执行任务，这样就避免了频繁创建/销毁线程所需的开销。然而由于我学艺不精，并不知道具体如何实现一个线程池（但是想必存在开源的已经写好的线程池库）。当然，也可以在主线程中准备数据，每准备好一部分就开启一个线程进行处理，最后再等待所有线程结束后收集好计算结果。但是要注意，最大线程数量是有限制的，请合理划分数据。

问：你这多线程编程自己都不会，还出来写指南？
答：正常小朋友一般问不出来这种问题。

第三章　第一部分结束语

在这一部分，我们借助 c 语言了解了 SIMD 和多线程编程的基本知识，并通过示例程序展示了并行计算带来的性能提升。
希望这一部分能对您有所帮助。

最后的最后，我们来解答这一部分中最神秘的问题：**为什么要计算 a×b+c**。其实，是为了介绍一个非常有用的指令——FMA。由于这份指南是关于并行计算的，所以关于 FMA 的事情就放在角落里了。
FMA —— Fused Multiply-Add，称作 “积和熔加” 运算。该运算直接计算 a*b+c 的值。其实这条指令的目的并不是加速计算（虽然好像的确比先乘后加快），其目的在于提高精度。众所周知，浮点数并不是无限精度的（可以参考附录中的浮点数的机器表示方法）。先乘后加包含了两次近似，而 “积和熔加” 只进行一次近似。但是要注意，这条指令也不是万能的。维基百科上有举例：计算 $x^2 - y^2$，如果 $x = y$，而程序写成 `FMA(x,x,-y\*y)`, 那么 `y\*y` 会先进性近似，之后可能会与 `x\*x` 的精确值不相等，从而得到一个非 0 的结果。如果进行多步运算，误差也许会累积到后面，甚至逐渐放大。不过一般来说，FMA 还是一个好指令。在 c 语言中，math.h 定义了 fma 函数用于做 “积和熔加” 运算。具体的汇编语言怎么写就不在这里详述了。

第二部分

GPGPU

第四章　GPU 简介

GPU，即 Graphics Processing Unit，顾名思义，是用来处理图形的。那它可能就要问了，我明明是用来处理图形的，怎么就跑来做通用计算了呢？卡的一生啊，不可预料……

4.1　图形处理是怎么回事？

色不异空，空不异色；色即是空，空即是色。——《般若波罗蜜多心经》

首先来简单解一下图形处理的本质是什么。
来看看我们在玩游戏的时候，立体图像是怎么显示到显示器上的。首先，立体模型是以三角形的方式组织的。把一个立体模型的表面分成一个一个的三角形，把三角形们的顶点和法向量组织起来，就可以表示一个模型了。使用的三角形越小，模型就越细腻。为了把这个模型显示出来，我们先拿到所有顶点的坐标，根据从屏幕看过去的角度和举例将顶点做坐标变换。然后根据顶点的连接情况把顶点变成三角形，算出投影在屏幕上的坐标。变换完再根据环境光照、物体材质等等信息，把三角形涂上颜色。（这只是大致的工作流程，可能与实际情况不完全相同，但是有那个意思。）

没什么感觉？再仔细想想，是不是对于每个顶点的处理来说，完全不用关心其他顶点的情况？而且对于每个顶点的处理其实都是非常简单的运算（坐标变换什么的）。于是 GPU 就这么被设计出来了。

4.2　那 GPGPU 是啥？

一个人的命运啊，当然要靠自我奋斗，但是也要考虑到历史的行程。——他

GPU 本来就是为了绘图而设计的。但是某一天，它的计算能力超过 CPU 了！主要是 GPU 的核心数量越来越多，虽然单个核心和 CPU 比起来要菜不少，但是人家核多啊！于是，这种算得快的能力被利用起来，搞出了 GPGPU，也就是 General-Purpose computing on Graphics Processing Units，让显卡来做通用计算。

比较完善的 GPU 编程库（库?）有 CUDA 和 OpenCL。OpenCL 可以运行在 CPU、GPU 甚至 FPGA 之类的设备上，而 CUDA 只能运行在 Nvidia 的显卡上。然而现今 CUDA 发展远好于 OpenCL。

我个人感觉 CUDA 用起来要简单一些。

第五章　CUDA

CUDA (Compute Unified Device Architecture) 是 Nvidia 推出的用于自家显卡的并行计算技术。这里将介绍如何使用 CUDA 来进行并行运算。

5.1　开始之前

工欲善其事，必先利其器。——《论语》

CUDA 需要 gcc 版本高于 5，推荐的版本是 9.x。服务器上已经安装了 gcc9.3.0，可以直接 load

```
module load gcc/9.3.0
```

之后，要装载 cuda 的 module

```
module load cuda/11.0.3
```

这样，nvcc 编译器和各种库就添加到环境变量里面了。

5.2　我的第一个 cuda 程序

信じる心があなたの魔法！——『リトルウィッチアカデミア』

还是之前的先乘后加。我们把计算过程放到 GPU 上去做。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>

__global__ void muladd(double* a, double* b, double* c, double* d,
                       unsigned int N){
    unsigned int id = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int j;
    if(id < N){
        for(j = 0; j < 1000000; j++){
            d[id] = a[id] * b[id] + c[id];
        }
    }
}

int main(){
    double* a;
    double* b;
    double* c;
    double* d;

    double* cua;
    double* cub;
    double* cuc;
    double* cud;

    a = (double*)(malloc(8192*sizeof(double)));
    b = (double*)(malloc(8192*sizeof(double)));
    c = (double*)(malloc(8192*sizeof(double)));
    d = (double*)(malloc(8192*sizeof(double)));

    cudaMalloc(&cua, 8192*sizeof(double));
```

```
    cudaMalloc(&cub, 8192*sizeof(double));
    cudaMalloc(&cuc, 8192*sizeof(double));
    cudaMalloc(&cud, 8192*sizeof(double));

    //Prepare data
    unsigned long long i;
    for(i = 0; i < 8192; i++){
        a[i] = (double)(rand()%2000) / 200.0;
        b[i] = (double)(rand()%2000) / 200.0;
        c[i] = ((double)i)/10000.0;
    }

    clock_t start, stop;
    double elapsed;
    start = clock();
    cudaMemcpy(cua, a, 8192*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(cub, b, 8192*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(cuc, c, 8192*sizeof(double), cudaMemcpyHostToDevice);

    muladd<<<32, 256>>>(cua, cub, cuc, cud, 8192);

    cudaMemcpy(d, cud, 8192*sizeof(double), cudaMemcpyDeviceToHost);
    stop = clock();
    elapsed = (double)(stop-start) / CLOCKS_PER_SEC;
    printf("Elapsed time = %8.6f s\n", elapsed);
    for(i = 0; i < 8192; i++){
        if(i % 1001 == 0){
            printf("%5llu: %16.8f * %16.8f + %16.8f = %16.8f (%d)\n",
                   i, a[i], b[i], c[i], d[i], d[i]==a[i]*b[i]+c[i]);
        }
    }

    free(a);
```

```
    free(b);
    free(c);
    free(d);
    cudaFree(cua);
    cudaFree(cub);
    cudaFree(cuc);
    cudaFree(cud);
}
```

编译 cuda 程序要使用 `nvcc` 编译器。

```
  $ nvcc -fmad=false  -o cumuladd muladd.cu
```

注意这里的`-fmad=false`。nvcc 默认是会把相邻的乘法和加法优化成 FMA 的（关于 FMA，参见上一部分最后一章 (三)）。这里关闭 FMA 来跟 CPU 同台竞技，也为了后面比较结果时不会出差错。

运行一下，哇！比 1s 还短！

（其实 nvcc 只负责 GPU 部分，剩下的会交给普通的 c++ 编译器。不好意思，CUDA runtime API 是 c++ 的，但是众所周知，c++（基本上）和 c 是兼容的。所以本质上上面是一个看起来像 c 的 c++ 程序 ）

我们来解释一下这里都干了些啥

5.2.1 #include <cuda_runtime.h>

大多数我们需要的功能都在 cuda_runtime.h 头文件里提供。要 include 进来哦！

5.2.2 `__global__ void muladd`

注意 `__global__`，这是 CUDA 对 c++ 的扩展。常见的标签：

- `__global__`,
- `__device__`, 和

- `__host__`
- 还有别的不常用的，可以参考 cuda runtime 文档

`__global__` 的函数可以被 CPU 或 GPU 调用，在 GPU 上执行。（在 GPU 上调用 `__global__` 函数似乎是在并行地运行并行程序（就是套娃 ）。一般还是从 CPU 调用比较常见。）这种函数被成为 “kernel”。

`__device__` 的函数只可以被 GPU 调用，在 GPU 上执行。这种函数一般会被编译器展开到调用处。

`__host__` 的函数只可以被 CPU 调用，在 CPU 上执行。和不加标签一样。

`__global__` 函数必须是返回值为 `void` 的函数，`__device__` 和 `__host__` 则无要求。

5.2.3 blockIdx,blockDim 和 threadIdx

一个 “kernel” 事实上要同时被许多核心执行，那么每个核心就需要知道自己处理的是哪个数据，要是大家抢同一个数据的话就没有意义了。CUDA 设计了两级的网格——grid 和 block。每个 grid 可以包含若干个 block，每个 block 又包含若干个 thread。一个 thread 将会在一个核心上执行。但是要注意，每个 block 包含的 thread 数目不能超过一个最大值（应该是 1024）。此外，根据程序使用的寄存器数目，一个 block 可以包含的 thread 数目可能会再小一点。（GPU 的寄存器还是很多的，一般来说不用担心）此外，每个 block 里的 thread 数目最好设为 32 的倍数，其中原理可以参考后续章节。

这里的 blockIdx，blockDim 和 threadIdx 就相当于线程的编号，用于知道自己应该取处理哪些数据。blockIdx 就是 block 在 grid 中的编号，blockDim 是 block 中 thread 的数目，threadIdx 是 thread 在 block 中的编号。

注意到后面的`.x`, CUDA 允许使用最大三维的 block 和三维的 grid。上面的程序只使用了一个维度，就只需要`.x` 即可。

下图展示了 grid，block 和 thread 的关系。

事实上也有 gridDim，用来表示 grid 在各个维度上包含多少 block。

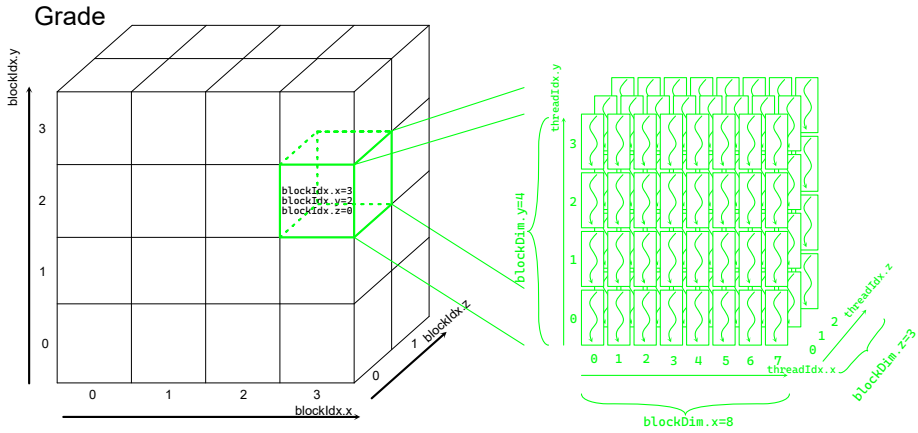


图 5.1: grid, block 和 thread

5.2.4 cudaMalloc

类似于 `malloc`，但是是在显存中申请空间。第一个参数要传入一个指针的指针，函数执行完后第一个参数所指向的指针就是显存空间的指针了。第二个参数就是以 byte 计的空间大小。

5.2.5 cudaMemcpy

用于系统内存和显存之间进行数据拷贝。第一个参数是目标指针，第二个参数是源头指针，第三个参数是要拷贝的内容的大小（byte 单位），第四个参数设置拷贝方向。

为什么要有第四个参数呢？其实指针类型相当于是整数，并没有标识这是系统内存的指针还是显存的指针，所以需要第四个参数说明拷贝方向。

第四个参数有一下几种可选值：

- `cudaMemcpyHostToHost`

- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`

具体作用从名字上看就很明显了。

5.2.6 <<<32, 256>>>

三重尖括号也是 CUDA 对 C++ 的扩展，用来表示以何种方式组织 grid 和 block。原则上 `<<<>>>` 里面应该接受两个 dim3 类型的量。

> dim3 是基于 uint3 的类型，uint3 是 CUDA 定义的一种矢量类型，顾名思义，就是三个 unsigned int 放在一起。（事实上，就是一块 12 字节的连续空间而已）。定义 dim3 变量时，未指定的维度上的数会被设置为 1.
> 所以上面的 `<<<>>>` 里面直接写两个数字其实代表了 x 维度，剩下两个维度自动设置成 1 了。

里面第一个参数是对 grid 的描述，第二个是对 block 的描述。比如要使用如图 6.1 的结构运行一个叫做 myKernel 的 `__global__` 函数：（假设该函数不需要参数）

```
dim3 blocks(4, 4, 2); // grid包含4*4*2个block
dim3 threadsPerBlock(8, 4, 3); // 每个block包含8*4*3个thread
myKernel<<<blocks, threadsPerBlock>>>();
```

5.2.7 cudaFree

`cudaFree` 是用来释放显存的，和 c 的 `free` 类似。
及时释放使用完的内存是好习惯哦

5.3 CUDA 新增类型

Be not afeard. The isle is full of noises, sounds and sweet airs, that give delight and hurt not.——The Tempest

除了上面提到的 `dim3` 之外，CUDA 还提供了其他一些矢量类型，如 double2，double3，double4，float2，float3，float4 还有 char/short/int/long/long long 及其无符号系列。具体可以参考 cuda vector types[1]。但是，这些矢量类型并不支持 SIMD，甚至没有实现矢量加减法，可以把它们看成只是为了方便表示矢量而定义的类型，实际使用的时候还需自己拿每个分量去做计算。

另外，CUDA 还支持一种半精度浮点数，每个 16bit（也有相应的矢量类型）。（关于计算机里的浮点数表示可以参考附录）
这种半精度类型可能在机器学习领域比较有用。根据 CUDA 的文档，使用半精度的话建议用 `half2` 这种矢量类型（一个 `half2` 是一个两维的 `half` 矢量），因为 `half2` 可以用一些特别的函数（如 `__hadd2`（矢量加法）之类的，还有乘法、除法等等）在一个指令中做两个加法。这可能是 CUDA 唯一的 SIMD 操作吧。

5.4 使用 `__device__` 函数

不在其位，不谋其政——《论语》

一般我们写一个程序都包含许多函数，它们调用来调用去，组合成我们想要的样子。使用函数的好处在于重复的代码可以只写一次。而且当这部分功能需要改变的时候只要修改函数内容就可以了。

在 CUDA 里我们也可以这么做。下面我们分别实现一个乘法函数和一个加法函数，让 kernel 调用它们完成计算。

```
__device__ double myAdd(double a, double b){
  return a+b;
}

__device__ void myMul(double* a, double* b, double* c){
```

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#vector-types

```
  *c = *a + *b;
}

__global__ void muladd(double* a, double* b, double* c, double* d,
                       unsigned long long N){
    unsigned long long id = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned long long j;
    if(id < N){
        for(j = 0; j < 1000000; j++){
            myMul(a+i, b+i, d+i);
            d[i] = myAdd(c[i], d[i]);
        }
    }
}
```

这里的 myAdd 和 myMul 采用了不同的写法。myAdd 接受两个 double 参数，返回一个 double（如前文所述，__device__ 函数并不要求 void 返回值）。而 myMul 则是另一种风格，直接接受三个指向 double 的指针，把第三个指针指向的 double 设为前两个指针指向的 double 的和。

myAdd 和一般的函数设计思路是一致的。但是当你想要从一组参数获得多种返回值的时候，myMul 的设计思路可能会变得非常有用。当然，也可以使用引用传递来返回值：

```
__device__ void func(double a, double b, double &c){
    c = a * b;
}
```

__device__ 函数也支持递归，但我想不到什么需要并行且递归的运算。

5.5 生成动态链接库

生命在于静止。——蔡明

虽然这是一份讲解如何并行编程的指南，但是我想许多人分析数据不是纯粹使用

c/c++ 的。因此，使用动态链接库就很有意义了，这可以让你的并行程序安排到你常用的软件中，比如 root。（显然 root 的 cling 解释器是看不懂 cuda 代码的，cuda 也不大可能采用 cling 后端，因此创建动态链接库是有意义的。）

动态链接库在 linux 里面称作 “shared object”，后缀名是 “.so”.

5.5.1 用 c++ 做一个动态链接库

先来看看对于一个普通的 c++ 程序要怎么做。假设现在有一个 shared.h 头文件声明了一些函数，并且在 shared.cpp 中实现了这些函数。

```
$ g++ -shared -fPIC -o libshared.so shared.cpp
```

上面的代码会生成一个叫做 “shared.so” 的文件，这就是我们想要的动态链接库。

关于`-fPIC`:

> PIC 表示位置无关代码 (Position-Independent Code)。这里面的机制比较复杂，就不在这里详细说明了。但是一般来说这么用就对了。
> 似乎除非你的程序编译出来可执行文件大于 2GB 的话才需要考虑一些其他的特殊技术。

5.5.2 生成 cuda 的动态链接库

同样，假设在 cushared.h 里声明了一些函数，在 cushared.cu 里实现了这些函数（函数里包含对 kernel 的调用）。

```
$ nvcc --compiler-options '-fPIC' -o libcushared.so --shared mykernel.cu
```

`--compiler-options` 后面接着的是要传递给后端 c++ 编译器的参数。`--shared` 表示要编译成动态链接库。

5.5.3 使用动态链接库

在 c++ 中使用动态链接库有两种方法。一种是在代码里包含好头文件，并在编译的时候指定好要链接的库文件。例如我们之前的动态库 shared.cpp 和其头文件 shared.h. 假设我们有一个 call.cpp（和 shared.h 还有 libshared.so 在同一目录下），其中需要使用我们的 libshared.so 库，那么首先要在 call.cpp 里 `#include "shared.h"`. 然后用如下方法编译

```
$ g++ -o call -L. -lshared
```

其中，`-L.` 表明我们要把当前位置列入库文件搜索目录，`-lshared` 表示需要链接叫做 “shared” 的库。（`-lshared` 会搜索 libshared.so）。

但是这样运行 `./call` 会报错，这是因为系统并不知道这个 libshared.so 在哪里。所以我们应该把这个位置添加到环境变量 `LD_LIBRARY_PATH` 中。

```
$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

再执行 `./call` 就可以了。

另一种调用方法是使用 dlopen 函数直接打开库文件（比如 libshared.so），然后再通过 dlsym 函数找到需要的函数的地址，然后执行函数。用完以后还要用 dlclose 关闭库文件。这比较麻烦，就不多介绍了。

5.5.4 在 root 中交互地使用动态链接库

此方法在 *root6* 上运行成功，但是在 *root5* 中会失败。具体原因不明。

在 root 交互环境中，可以直接引入头文件，然后加载.so 文件，就可以调用函数了。

```
root [0] #include "shared.h"
root [1] .L libshared.so
root [2] //do something
```

5.5.5 在 root 中交互地使用 cuda 的完整例子

此方法在 *root6* 上运行成功，但是在 *root5* 中会失败。具体原因不明。

5.5.5.1 sharedlib.h

```
#ifndef SHARED_LIB_H
#define SHARED_LIB_H
void vecadd(double*, double*, double*, unsigned int)
#endif
```

5.5.5.2 sharedlib.cu

```
#include <cuda_runtime.h>
#include "./sharedlib.h"

__global__ void cuadd(double* a, double* b, double* c, unsigned int N){
    unsigned int id = blockIdx.x * blockDim.x + threadIdx.x;
    if(id < N){
        c[id] = a[id] + b[id];
    }
}

void vecadd(double* a, double* b, double* c, unsigned int N){
    double* cua;
    double* cub;
    double* cuc;
    cudaMalloc(&cua, N*sizeof(double));
    cudaMalloc(&cub, N*sizeof(double));
    cudaMalloc(&cuc, N*sizeof(double));
    cudaMemcpy(cua, a, N*sizeof(double), cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(cub, b, N*sizeof(double), cudaMemcpyHostToDevice);
    cuadd<<<(N+127)/128,128>>>(cua,cub,cuc,N);
    cudaMemcpy(c, cuc, N*sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(cua);
    cudaFree(cub);
    cudaFree(cuc);
}
```

5.5.5.3 compile

```
nvcc --compiler-options '-fPIC' -o libcusharedlib.so --shared share.cu
```

5.5.5.4 use

```
$ root
root [0] #include "sharedlib.h"
root [1] .L libcusharedlib.so
root [2] double* a = (double*)malloc(1024*sizeof(double))
root [3] for(int i = 0; i < 1024; i++){ a[i] = (double)i; }
root [4] double* c = (double*)malloc(1024*sizeof(double))
root [5] vecadd(a, a, c, 1024)
root [6] c[42] // 84.0000
root [7] free(a)
root [8] free(b)
root [9] .q
```

5.5.6 不支持 c++ 的情况

一些编程语言不支持调用 c++ 写成的库，只支持 c 的（以及在需要 c 调用 c++ 的库的情况）。这时候要对需要导出的函数做一点点处理。要将需要导出的函数声明放到 `extern "C"` 里面。比如在头文件里

```
extern "C"{
  int someIntFunction();
  double someDoubleFunction();
  //......
}
```

5.6 本章小结

感觉怎么样？有点似懂非懂？要不尝试翻回来再看看代码，自己改一改试一试，有没有觉得稍微明白一点了？可以试着改一改 thread 和 block 的结构，看看运行效率有怎样的变化。

还有要注意一点，这里程序比较简单，所以完全没有安排错误处理。`cudaMalloc`，`cudaMemcpy` 等函数都是有返回值的，会返回一个 `cudaError_t` 的类型。大家完全可以用 `auto` 类型来接收返回值。`cudaError_t` 是一个枚举类型，其实跟 int 没太大区别。当返回值是 `cudaSuccess`（这个等于 0，可以直接把返回值和 0 作比较判断有没有 error）的时候说明成功了，没有错误。其他各种错误可以参考 cudaRuntimeAPI[2]

此外，kernel 函数也有可能运行不成功，比如因为占用寄存器数量过多且每个 block 里的线程数比较多，可能会导致 kernel 没有成功执行。要处理这种错误，可以这样做：

```
some_kernel<<<10,1024>>>();
auto err = cudaGetLastError();
if(err != cudaSuccess){
```

[2] https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__TYPES.html#group__CUDART__TYPES_1g3f51e3575c2178246db0a94a430e0038

```
  //handle the error
}
```

喜闻乐见的自问自答环节！

问：为什么 thread 的数量是 32 的倍数比较好？

事实上，CUDA 搞的是一种 SIMT 模型（single instruction multiple threads），也就是说，一些线程是共享一条指令的。目前的所有支持 CUDA 的显卡都是以 32 个线程为一组（称作一个线程束（warp）），它们共同接受同一个指令并进行计算。如果一个 block 里的 thread 数目不是 32 的倍数，假设是 48，那么显卡会把这 48 个线程分成两组，一组 32 线程，一组 16 线程。在执行 16 线程的一组的时候，只有 16 个线程在干活，另外 16 个线程只能站在干岸上看着，形成一方有难八方围观之势。为什么这 16 个线程不能干点别的呢？因为硬件上就是把 32 个线程作为一组的来调度的。基于同样的原因，kernel 里面要尽量避免不必要的条件判断。因为每次发送指令，接受指令的线程都要干一样的事情，所以在条件语句中，是一个分支的线程执行完，再另一个分支的线程执行。如果两个分支的跳转概率一样，可能就会使得性能减半。

问：三维 grid 和三维 block 意义何在？

事实上，grid 和 block 的维度对于计算机来说并没有太大的意义，但是对于人来说还是有意义的。使用什么样的 grid 和 block 结构取决于数据的“样子”。

就像一维数组和三维数组一样，原则上，用多维数组处理的问题都可以转化到用一维数组来处理。

> 比如 10*10 的数组，访问起来是 `array2[i][j]`
> 换成一维数组呢，就可以这样访问：`array[i*10+j]`

问：二维和三维的 block 里面 thread 数量是怎么限制的？

不考虑寄存器数量限制的话：

首先，一个 block 里面不能有超过 1024 个线程。其次，在每个维度上都有一个最大线程数。只有同时满足以上限制才可以。（第一个维度，也就是 x 的最大值肯定是 1024，这保证了一维的 block 可以用满线程数限制）。

具体的最大值，还有好多其他参数，包括寄存器数量等等都有相应的函数可以查询。具体可以参考 cuda 的 sample 里面的 `1_Utilities/deviceQuery` 里面的程序以及运行结果。

问：我是使用 printf 法来 debug 的那种选手怎么办？
没关系，kernel 函数里可以使用 printf。但是要注意，在 kernel 里用 printf 的时候不要把 grid 和 block 设置得太大，否则你会得到铺天盖地的打印信息。（也可以在 kernel 里面判断，只有在特定 block 的特定 thread 打印信息也是可行的）

问：很好，那么这到底有什么用呢？
答：正常小朋友一般问不出来这种问题。

再补充一个信息（可能管理员会发现比较有用）。可以使用 `nvidia-smi` 查看 GPU 的工作情况。这个运行一下只会显示一次信息。如果相要持续显示信息，可以使用`-l` 或者`-lms` 选项。具体见 `nvidia-smi -h` 输出的帮助信息。

第六章　CUDA 高级优化技巧

这一章虽然称为高级优化技巧，其实不算什么高级内容，只是介绍一些有用的特性。

6.1　纹理和纹理内存

> 余忆童稚时，能张目对日，明察秋毫。见藐小微物，必细察其纹理，故时有物外之趣。——《浮生六记》

6.1.1　什么是纹理？

在图形学中，纹理本来是一幅图像，用来被 “贴” 在要显示的位置。由于观察角度不同，显示的点到纹理上的点会有不同的映射关系。

从纹理中取得信息和从数组中取得值是类似的，但是从纹理中取值被称作 “拾取”(fetch)，取到的内容称作 “texel”。

但是和从数组中取值不同，拾取纹理可以使用非整数的下标，甚至可以自动进行插值。

CUDA 支持整数纹理和单精度浮点数纹理以及相应的 2 维或 4 维矢量纹理。（这里说的是纹理里面的值，相当于数组的类型是整数/单精度浮点数或者它们对应的矢量类型）。

6.1.2　CUDA 纹理的寻址模式和线性插值

CUDA 的纹理支持 “归一化”，即将坐标和/或值线性映射到零到一的区间里。

CUDA 纹理支持整数拾取和浮点数拾取。同时对超过边界的拾取可设定为 “环绕”(wrap)，“钳制”(clamp)，“镜像”（mirror）或者 “边界”（border）。几种边界模式的不同如下（假设是 1 维的情况，纹理里的内容是 (1, 2, 3, 4)，只列出从-4 到 7 的 “整数点” 的值）

- 环绕：(1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
- 钳制：(1, 1, 1, 1, 1, 2, 3, 4, 4, 4, 4, 4)
- 镜像：(4, 3, 2, 1, 1, 2, 3, 4, 4, 3, 2, 1)
- 边界：(0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0)

  这里 “整数点” 加引号是由于纹素的坐标其实是有 0.5 的偏移的。用数组来举例：
  如果有数组 a，a[0]=0, a[1]=1, a[2]=2, a[3]=3
  那么，同样的纹理 t，t[0.5]=0, t[1.5]=1, t[2.5]=2, t[3,5]=3
  可以理解为类似把一个像素变成一个单位正方形，正方形的中心坐标和编号比起来是有 0.5 的偏移的。

注意，环绕和镜像模式只能在坐标 “归一化” 的的纹理中使用。

CUDA 纹理支持两种过滤模式，“最近点” 和 “线性插值”。顾名思义，最近点取坐标最近的格点的值，而线性插值则是用相邻的格点的值进行线性插值来作为拾取到的值。这里，“最近点” 用的是左闭右开区间，即 t[0]=t[0.5]=t[0.99],t[1]=t[1.5] 等等。

注意！线性插值的精度是有限的。内部是用一个 8bit 的数做插值 (我理解是把一个格子分成 256 份的意思)。所以如果对精度要求比较高，建议用最近点模式然后手动插值。如果对精度要求不是特别高的话，用线性插值模式可以获得更高的性能。

注意！纹理拾取函数接受的参数是单精度浮点数。如果坐标是双精度浮点数的话会自动转换成单精度。但是在最近点模式中，双精度转单精度并不是截断的，而是舍入到最近的单精度浮点，这意味着有可能因为进位而导致拾取到不同的值，这在手动插值的情况下会引发问题。建议手动转换至单精度或向下取整后传入参数。（建议使用 `__double2float_rd(double)` 手动向转换精度。这是由于向下取整得到的仍然是 double，在传参时还是会经历一次精度转换，可能会有一定的性能损失。）

6.1.3 举一个例子

其实是两个例子 ，分别是一维（和二维差不多）和三维的单精度浮点数纹理的使用。其余情况大同小异。

6.1.3.1 一维纹理的情况

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

__global__ void FetchFrom1DTexture(cudaTextureObject_t tex,
                                   float* position,
                                   float* result,
                                   unsigned int N){
    unsigned int id = blockIdx.x * blockDim.x + threadIdx.x;
    if(id < N){
        //拾取
        //<float>是c++模板
        //这里不详细解释模板了，总之在尖括号里填写纹理的类型就对了
        //第一个参数是一个texture object (cudaTextureObject_t)
        //第二个参数是要拾取的位置。
        result[id] = tex1D<float>(tex, position[id]);
    }
}

int main(){
    //我们要使用的纹理的原始值
    float* resource = (float*)(malloc(4*sizeof(float)));
    resource[0] = 1.0;
    resource[1] = 2.0;
    resource[2] = 3.0;
    resource[3] = 4.0;
```

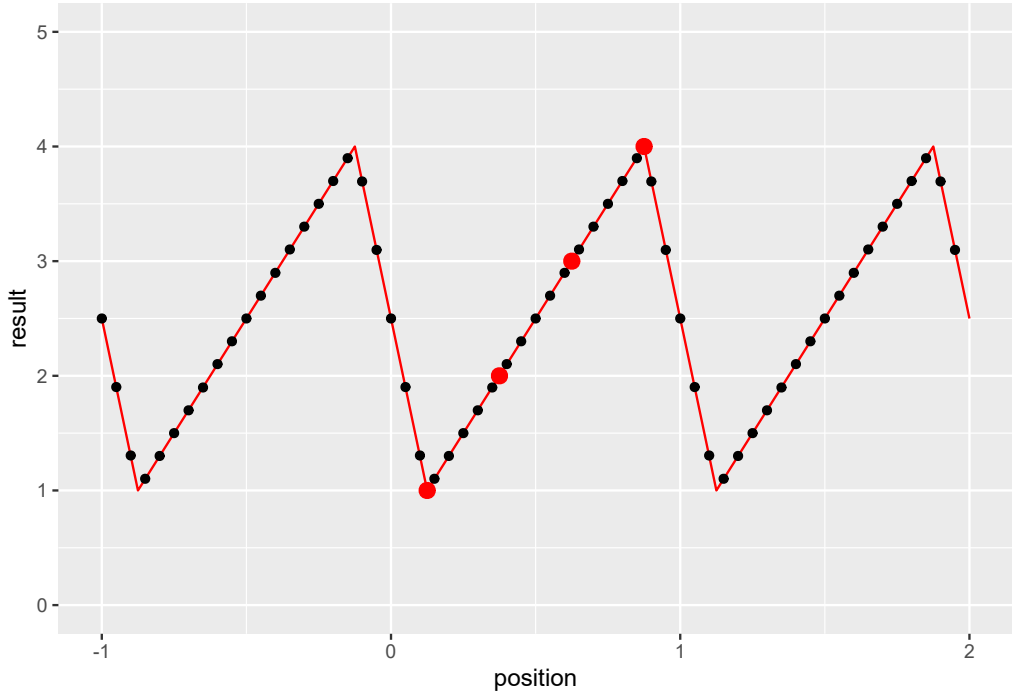
```
//创建一个CUDA Array

//cudaChannelFormatDesc:
//  描述Array里每个元素的样子。这里每个元素是float。
cudaChannelFormatDesc floatChannelDesc
                    = cudaCreateChannelDesc<float>();
//声明CUDA Array
cudaArray_t cuArray;

//为cuarray分配空间。第三个参数是array在第一个维度上的长度。
//这里长度单位是“个”，
//因为在floatChannelDesc里已经描述了每个元素的大小了。
//cudaMallocArray可以分配一维或二维的array
//第四个参数用于指定第二个维度的长度（单位是多少行(háng)），默认为0
cudaMallocArray(&cuArray, &floatChannelDesc, 4);

//向CUDA Array中拷贝数据。
//cudaMemcpy2DToArray用于向一维和二维Array中拷贝数据。
//一维Array就是第二个维度长度为0的二维Array
//第一个参数是目标CUDA Array
//第二个参数是目标的x位置
//第三个参数是目标的y位置
//第二个参数和第三个参数的意义在于可以只更新Array的一部分
//第四个参数是被拷贝的数据的指针
//第五个参数是描述被拷贝的内容按照每一行多少byte来排列
//第六个参数是目标位置每一行拷贝多少byte
//第七个参数是一共拷贝多少行
//第八个参数是拷贝方向，这里是从主机内存到显卡内存。原因前面讲过。
//可以通过后文的图片详细了解这个函数的工作方式。
cudaMemcpy2DToArray(cuArray, 0, 0,  resource, 4*sizeof(float),
                    4*sizeof(float), 1, cudaMemcpyHostToDevice);
//其实拷贝完以后主机内存里的resource已经没用了，可以现在就free掉。
```

```
//描述要成为纹理的东西是个啥
//resType设置为cudaResourceTypeArray，表明要从一个Array来构建纹理
//res.array.array设置成我们刚刚准备好的Array
struct cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeArray;
resDesc.res.array.array = cuArray;

//描述我们要的纹理是个啥样子
struct cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
//addressMode可取的值有：
//  cudaAddressModeWrap
//  cudaAddressModeClamp
//  cudaAddressModeMirror
//  cudaAddressModeBorder
//详情见上文描述
//addressMode是长度为3的数组，分别对应三个维度。
//这里是一维所以只用第一个。
texDesc.addressMode[0]   = cudaAddressModeWrap;
//filterMode可取的值有：
//  cudaFilterModeLinear  线性插值模式
//  cudaFilterModePoint   取最近点模式
texDesc.filterMode       = cudaFilterModeLinear;
//readMode可取的值有：
//  cudaReadModeElementType  Array是什么类型拾取出来就是什么类型
//  cudaReadModeNormalizedFloat  进行归一化
//注意，只有8位或16位整数支持归一化，其他类型比如int/float则不支持。
texDesc.readMode         = cudaReadModeElementType;
//是否对坐标归一化到[0,1). 1表示进行归一化。
texDesc.normalizedCoords = 1;

//声明并创建纹理
```

```
//这里相当于将纹理绑定到之前创建的cuArray
//所以不能释放掉cuArray
cudaTextureObject_t tex1DObj;
cudaCreateTextureObject(&tex1DObj, &resDesc, &texDesc, NULL);

//我们想要拾取的纹理的坐标。
float* position = (float*)(malloc(60*sizeof(float)));
for(int i = 0; i < 60; i++){
    position[i] = -1.0 + (float)(i)/20.0;
}

float* result = (float*)(malloc(60*sizeof(float)));

float* cuposition;
float* curesult;
cudaMalloc(&cuposition, 60*sizeof(float));
cudaMalloc(&curesult, 60*sizeof(float));
cudaMemcpy(cuposition, position, 60*sizeof(float),
           cudaMemcpyHostToDevice);

FetchFrom1DTexture<<<1, 60>>>(tex1DObj, cuposition, curesult, 60);

cudaMemcpy(result, curesult, 60*sizeof(float),
           cudaMemcpyDeviceToHost);

for(int i = 0; i < 60; i++){
    printf("Texel at position (%- 5.3f) is %5.3f \n",
           position[i], result[i]);
}

cudaFree(curesult);
cudaFree(cuposition);
```

```
    free(result);
    free(position);
    free(resource);

    //不再使用的纹理要销毁掉
    cudaDestroyTextureObject(tex1DObj);
    //Array用完了也要释放
    cudaFreeArray(cuArray);

}
```

画图来表示一下结果的话



上图中红点是我们设定的值，由于设置了 `texDesc.normalizedCoords = 1;`，[0,4) 被映射到了 [0,1)，相应的位置 {0.5, 1.5, 2.5, 3.5} 就被映射到了 {1/8, 3/8, 5/8, 7/8}。由于设置了 `texDesc.addressMode[0] = cudaAddressModeWrap;`，超过 [0,1) 范围的位置得到了从 [0,1) 平移过去的值。

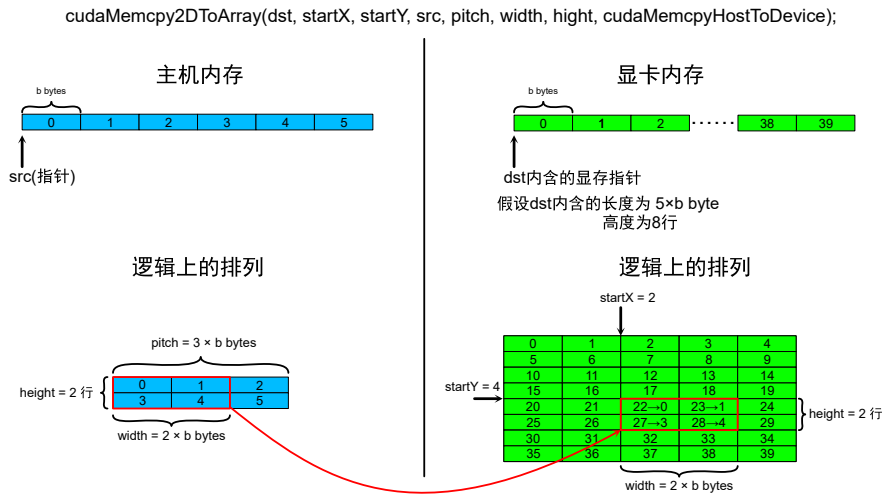


图 6.1: 图解 cudaMemcpy2DToArray

二维纹理与上述过程大同小异。

6.1.3.2 三维纹理的情况

三维纹理则有所不同。在这个例子里面，我们使用 float2 作为纹理的类型。

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

__global__ void FetchFrom1DTexture(cudaTextureObject_t tex,
                                   float3* position,
                                   float2* result,
                                   unsigned int N){
    unsigned int id = blockIdx.x * blockDim.x + threadIdx.x;
    if(id < N){
        //拾取
        //tex3D的后三个参数分别为第一，第二和第三个维度上的坐标
        result[id] = tex3D<float2>(tex, position[id].x,
```

```
                                        position[id].y,
                                        position[id].z);
    }
}

int main(){
    //我们要使用的纹理的原始值
    float2* resource = (float2*)(malloc(8*sizeof(float2)));
    for(int i = 0; i < 8; i++){
        resource[i].x = 1.0 + (float)i;
        resource[i].y = -2.0 - 2.0 * (float)i;
        //顺便一提:
        //float2的两个分量是x和y
        //float3的三个分量是x, y和z
        //float4的四个分量是x, y, z和w
    }

    //创建一个CUDA Array
    //cudaChannelFormatDesc:
    //  描述Array里每个元素的样子。这里每个元素是float2。
    cudaChannelFormatDesc floatChannelDesc
                        = cudaCreateChannelDesc<float2>();

    //cudaExtent是用来描述array的形状的
    //三个参数分别是宽(x), 高(y), 和深(z)
    cudaExtent ext = make_cudaExtent(2,2,2);
    //声明CUDA Array
    cudaArray_t cuArray;
    //为cuArray分配空间。
    cudaMalloc3DArray(&cuArray, &floatChannelDesc, ext);

    //由于三维拷贝参数比较复杂
    //所以CUDA设计了一个类型用于表达参数
```

```
cudaMemcpy3DParms cpy3d={0};
//纹理来源于resource数组
cpy3d.srcPtr.ptr = resource;
//来源数组中每两个float2算作一行
cpy3d.srcPtr.pitch = 2*sizeof(float2);
//来源数组每行取两个元素
cpy3d.srcPtr.xsize = 2;
//来源数组每两行组成一层
cpy3d.srcPtr.ysize = 2;
//拷贝的目的地是cuArray数组
cpy3d.dstArray = cuArray;
//目标里面在三个维度上各有几个元素，也是用cudaExtent
//这里要复制到整个array，所以重用了前面定义的ext
cpy3d.extent = ext;
//从主机内存拷贝到显存
cpy3d.kind = cudaMemcpyHostToDevice;
//三维拷贝也可以像二维拷贝一样设置起点，这里就不详述了
//可以参考cuda Toolkit 11.0.3的文档里
//cuda runtime API中5.9节关于cudaMemcpy3D的说明
cudaMemcpy3D(&cpy3d);


struct cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeArray;
resDesc.res.array.array = cuArray;

struct cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
//这回我们使用“边界”模式
//因为是三维纹理，所以三个都要设置
texDesc.addressMode[0]   = cudaAddressModeBorder;
texDesc.addressMode[1]   = cudaAddressModeBorder;
```

```
    texDesc.addressMode[2]   = cudaAddressModeBorder;

    texDesc.filterMode       = cudaFilterModeLinear;
    texDesc.readMode         = cudaReadModeElementType;
    //这次不做归一化
    texDesc.normalizedCoords = 0;

    //声明并创建纹理
    cudaTextureObject_t tex3DObj;
    cudaCreateTextureObject(&tex3DObj, &resDesc, &texDesc, NULL);

    //我们想要拾取的纹理的坐标。
    float3* position = (float3*)(malloc(64*sizeof(float3)));
    for(int i = 0; i < 4; i++){
        for(int j = 0; j < 4; j++){
            for(int k = 0; k < 4; k++){
                int id = i + j*4 + k*16;
                position[id].x = (float)i - 0.5;
                position[id].y = (float)j - 0.5;
                position[id].z = (float)k - 0.5;
            }
        }
    }

    float2* result = (float2*)(malloc(64*sizeof(float2)));
    float3* cuposition;
    float2* curesult;
    cudaMalloc(&cuposition, 64*sizeof(float3));
    cudaMalloc(&curesult, 64*sizeof(float2));
    cudaMemcpy(cuposition, position, 64*sizeof(float3),
               cudaMemcpyHostToDevice);

    FetchFrom1DTexture<<<1, 64>>>(tex3DObj, cuposition, curesult, 64);
```

```
    cudaMemcpy(result, curesult, 64*sizeof(float2),
               cudaMemcpyDeviceToHost);
    for(int i = 0; i < 4; i++){
        for(int j = 0; j < 4; j++){
            for(int k = 0; k < 4; k++){
                int id = i + j*4 + k*16;
                printf("Texel at position (%- 5.3f, %- 5.3f, %- 5.3f)"
                       " is (%- 5.3f, %- 5.3f) \n",
                        position[id].x, position[id].y, position[id].z,
                        result[id].x, result[id].y);
            }
        }
    }
    cudaFree(curesult);
    cudaFree(cuposition);
    free(result);
    free(position);
    free(resource);
    cudaDestroyTextureObject(tex3DObj);
    cudaFreeArray(cuArray);

}
```

可以尝试取不同的位置看看拾取到的值是什么。

6.1.4 如何实现双精度纹理

CUDA 不支持双精度纹理，但是可以用一个 int2 取去 “骗”CUDA。
用 `cudaCreateChannelDesc<int2>()` 生成 int2 的描述。
取到纹理以后可以用 `__hiloint2double` 将两个 int 转化为 double。
但是注意，这里必须得将 filterMode 设置为 `cudaFilterModePoint`，因为对两个 uint 插值再组合成 double 肯定和对 double 插值得到的结果不同。（详细原理可以参考附录中的浮点数的计算机表示方法）。

假设用 `int2 texel = tex3D<int2>(texture, position);` 拾取了 texel
要生成 double 需要 `__hiloint2double(texel.y, texel.x)`
这里先用 y 后用 x 似乎有点反直觉，但这其实是因为处理器是小端序的，因此要先写 y

6.1.5 所以为什么要用纹理？

上面提到了使用纹理的诸多限制，而且还很麻烦。
比如要使用 double 的话不能插值，还要额外进行转换。即便是可以插值的 float，插值精度也不如手动插值，那是不是这些情况下使用纹理有什么意义呢？
这涉及到缓存的原理。简单来讲，每次访问内存，如果要访问的内容没有被缓存，那么缓存会把要访问的内存和其附近的内存缓存起来，下次如果访问相邻的内存可以直接从缓存获取数据，减少访问内存的开销。而一般的缓存设计认为内存是线性的，但是在二维或者三维情况下，每次访问内存后，接着访问的很可能是逻辑上在空间中相邻的内存，这样就无法利用缓存进行高速读写。GPU 中有专门的纹理缓存，具有空间局部性，会缓存在逻辑空间中临近的内容，而不再是线性内存的“附近”。

CUDA 的文档中提到纹理缓存具有二维的空间局部性，但是没有提到三维的情况。因此不能确定缓存在三维纹理的情况下是如何工作的。但即便是二维的局部性，也会比普通的一维缓存命中率高一些。

6.2 流

参差荇菜，左右流之。窈窕淑女，寤寐求之。——《诗经》

之前的程序，我们都是 CPU 安置好数据以后把任务安排给 GPU，然后就开始等，等到 GPU 运行完程序以后把数据拿回来。但是我们是不是可以把 CPU 也利用起来干点别的？可以的！

这里要引入一个流的概念。我们可以创建一个流，然后把需要做的事情放进流里面，一个流里有任务的话就会自动开始工作。这时主机的工作只是把任务推到流里面，然后就可以做别的工作了。最后再让流和自己同步一下，保证流里的工作做完了即可。

事实上，当不创建新的流的时候，是使用默认流来工作的，这个流是“阻塞”的，也

就是说这个流里的任务完成之前会防止 CPU 进行下一步操作。而我们另外创建的流是“非阻塞”的，不会对 CPU 的工作流程造成影响。

6.2.1 举个例子

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>

__global__ void muladd(double* a, double* b, double* c, double* d,
                       unsigned long long N){
    unsigned long long id = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned long long j;
    if(id < N){
        for(j = 0; j < 1000000; j++){
            d[id] = a[id] * b[id] + c[id];
        }
    }
}

int main(){
    double* a;
    double* b;
    double* c;
    double* d;

    double** cua;
    double** cub;
    double** cuc;
    double** cud;

    cudaMallocHost(&a, 8192*sizeof(double));
```

```
cudaMallocHost(&b, 8192*sizeof(double));
cudaMallocHost(&c, 8192*sizeof(double));
cudaMallocHost(&d, 8192*sizeof(double));

cua = (double**)(malloc(2*sizeof(double*)));
cub = (double**)(malloc(2*sizeof(double*)));
cuc = (double**)(malloc(2*sizeof(double*)));
cud = (double**)(malloc(2*sizeof(double*)));

unsigned long long i;
for(i = 0; i < 8192; i++){
    a[i] = (double)(rand()%2000) / 200.0;
    b[i] = (double)(rand()%2000) / 200.0;
    c[i] = ((double)i)/10000.0;
}

//声明流
cudaStream_t streams[2];
//创建流，并分配一些显存空间。
for(int i = 0; i < 2; i++){
    cudaStreamCreate(&streams[i]);
    cudaMalloc(&(cua[i]),1024*sizeof(double));
    cudaMalloc(&(cub[i]),1024*sizeof(double));
    cudaMalloc(&(cuc[i]),1024*sizeof(double));
    cudaMalloc(&(cud[i]),1024*sizeof(double));
}

for(int i = 0; i < 8; i++){
    cudaMemcpyAsync(cua[i%2], a+i*1024, 1024*sizeof(double),
                    cudaMemcpyHostToDevice, streams[i%2]);

    cudaMemcpyAsync(cub[i%2], b+i*1024, 1024*sizeof(double),
                    cudaMemcpyHostToDevice, streams[i%2]);
```

```
        cudaMemcpyAsync(cuc[i%2], c+i*1024, 1024*sizeof(double),
                        cudaMemcpyHostToDevice, streams[i%2]);

        muladd<<<16, 256, 0, streams[i%2]>>>(cua[i%2], cub[i%2],
                                             cuc[i%2], cud[i%2], 1024);

        cudaMemcpyAsync(d+i*1024, cud[i%2], 1024*sizeof(double),
                        cudaMemcpyDeviceToHost, streams[i%2]);
    }
    //CPU现在可以干点别的
    //......
    //搞完以后
    //和显卡同步一下，保证显卡上的每个流都执行完了里面的任务
    cudaDeviceSynchronize();

    for(i = 0; i < 8192; i++){
        if(i % 1001 == 0){
            printf("%5llu: %16.8f * %16.8f + %16.8f = %16.8f (%d)\n",
                    i, a[i], b[i], c[i], d[i], d[i]==a[i]*b[i]+c[i]);
        }
    }

    for(int i = 0; i < 2; i++){
        cudaFree(cua[i]);
        cudaFree(cub[i]);
        cudaFree(cuc[i]);
        cudaFree(cud[i]);
        //用完的流要销毁掉
        cudaStreamDestroy(streams[i]);
    }
    //cudaFreeHost 和 cudaMallocHost相对应

    free(cua);
```

```
    free(cub);
    free(cuc);
    free(cud);

    cudaFreeHost(a);
    cudaFreeHost(b);
    cudaFreeHost(c);
    cudaFreeHost(d);
}
```

上面的程序创建了两个流。为了方便管理，每个流使用的显存空间也是分开申请的。

注意到程序里面分配主机内存空间的时候没有使用 `malloc` 而是使用了 `cudaMallocHost`。这是因为非同步拷贝需要被拷贝的主机内容是“页锁定”的内存，而直接用 `malloc` 申请的空间不具备这种性质。

> 内存是分页的，具体讲起来比较复杂，大致来说就是程序中使用的地址是虚拟的，然后在经过一个虚拟地址到真实地址的映射来访问真实内存位置。而真实位置甚至可能是不连续的。而所谓“页锁定”内存是真实内存地址，不分页的。这样 GPU 从内存读取数据就不需要通过 CPU 查询页表，可以直接从内存取数据了。使用 `cudaMallocHost` 不仅使得使用流成为可能，还可以提高数据传输效率。
> 但是要注意，不分页意味着即使内存占用过高的时候操作系统也不能将这部分内存放入交换空间。因此申请的页锁定内存过大可能会影响其他用户的体验。这些内存使用完后也请及时释放。

后面的 `cudaMemcpyAsync` 和 `cudaMemcpy` 是类似的，只不过要在最后一个参数里指定这个操作要被安排在哪一个流里面。

调用 kernel 的时候，`<<<...>>>` 里面的参数变成了 4 个。前两个还不变，第三个表示要动态申请的共享内存的大小，由于这里不需要共享内存，填 0 即可。第四个参数是要使用的流。

> 共享内存是一种“片上内存”，和显存空间是独立的。共享内存空间比较小，但是速度比显存快。共享内存是分配给 block 的，也就是说每个 block 能看到自己的共享内存，但是互相是看不到的。一个 block 中的所

有线程都可以看到这个 block 的共享内存，因此可以通过共享内存来通信。每一个块能拥有的共享内存大小可以查询（参考 CUDA sample 里的 1_Utilities/deviceQuery）。

后面的 `cudaDeviceSynchronize();` 可以同步一块显卡上的所有流。它会等待直到所有流里的所有任务都完成后才返回，相当于让 CPU 等待 GPU 完成工作。如果想要只同步一个流，可以用 `cudaStreamSynchronize` 函数，传入一个 `stream_t` 变量即可同步对应的流。

一个流里的任务是按照推入任务顺序完成的，流之间如果不进行同步是不保证执行顺序的。

6.2.2 回调

除了可以向流里添加内存拷贝任务和执行 kernel 任务，还可以添加回调任务。

回调任务相当于一个运行在 CPU 上的函数，当回调任务前面的任务都结束后，会自动调用这个函数。

下面的代码定义一个回调函数

```
void CUDART_CB my_callback(cudaStream_t stream,
                           cudaError_t status, void* data) {
    char* message = (char*)(data);
    printf("Callback!\n");
    printf("Message: %s\n", message);
}
```

其中的 data 是 `void` 型的指针，可以用来向回调函数传递信息。这里我们随便传递一个字符串。

下面的代码将上面的回调函数添加到一个叫做 `stream1` 的流

```
char* msg = "Encountering a callback!";
cudaStreamAddCallback(stream1, my_callback, (void*)msg, 0);
```

其中第四个参数是为未来的功能预留的，现在必须设为 0.

第三个参数就是即将变成上面定义回调函数时设定的 `void* data` 的内容。

6.2.3 什么时候要使用流

当数据拷贝和数据处理用时差不多的时候使用流可以提高速度。这是因为进行计算的时候内存和显卡之间的带宽时闲置的。使用多个流可以在计算一部分数据的时候拷贝其他数据。

同时，由于 PCIe 总线是全双工的，主机到设备的拷贝和设备到主机的拷贝也是可以同时进行的。

如果拷贝上花的时间比计算时间长，也许要考虑包含拷贝时间的话这种计算是不是真的比 CPU 快了。（其实拷贝还蛮快的）

如果是计算密集型的，拷贝时间可以忽略不计的话不建议使用非默认的流（除非希望在 CPU 上同时进行其他工作）。

此外，永远不建议将数据分为非常多的小块。因为拷贝数据的带宽虽然很宽，但是有一定的延迟。分太多的块的话延迟时间累积起来也不可小觑。

6.3 缓存和共享内存的分配

这一节的内容我也没有尝试过，但是原则上应该是可行的。

共享内存是似乎是直接放在高速缓存中的。cuda 允许调整共享内存和 L1 高速缓存的比例。

通过 `cudaFuncSetCacheConfig` 函数可以调整。函数的第一个参数是一个 kernel 函数（也就是 `__global__` 函数），第二个参数可以是

- cudaFuncCachePreferNone 使用默认值
- cudaFuncCachePreferShared 更多的共享内存
- cudaFuncCachePreferL1 更多的 L1 高速缓存
- cudaFuncCachePreferEqual 平均分配

6.4 本章小结

这一章就到这里了。其他内容比如各种 “局部内存”、“常量内存” 和 “共享内存” 等内容，还有其他各种~~奇技淫巧~~技术我也没有使用过，这里就不谈了。诸君感兴趣可以自己找资料看一看。

到了这一章，小朋友已经问不动问题了。

诸君有疑问再讨论吧。有合适的问题可以在后续版本中更新上来。

第七章　使用 Nsight 优化程序

前文提到了使用流增加并行性，但是如何判比较据拷贝和计算分别占用的时长呢？这里隆重介绍一下 Nsight。

Nsight 可以对一个程序运行期间设备的行为进行采样，并在一个时间轴上画出来。这样就可以方便地分析数据传输延迟是不是产生了比较大的影响，计算和数据拷贝是不是可以更加高效地并行等等。

Nsight 可以通过 SSH 连接到远程机器来监控程序运行情况，因此可以在本地安装 Nsight 来对服务器上的程序进行调试。

7.1　下载和安装

Nsight 要安装到自己的计算机中，然后连接到服务器进并分析服务器上程序的行为。

在这里[1]下载 nsight systems。下载后安装即可。

7.2　连接到服务器

如图，设置好服务器地址，ssh 端口和用户名。点击 connect 即可连结。

[1] https://developer.nvidia.com/nsight-systems

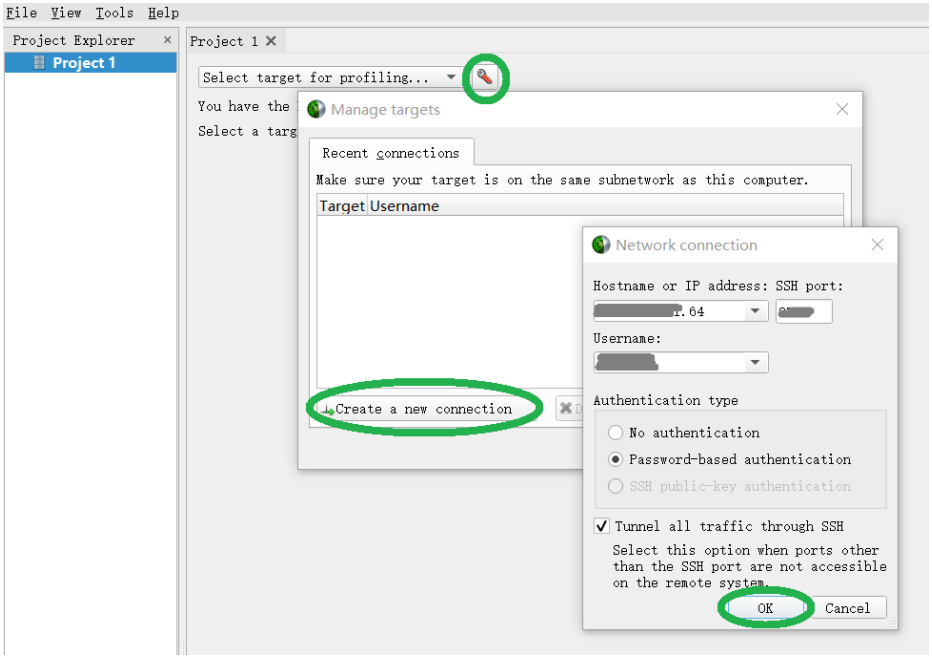


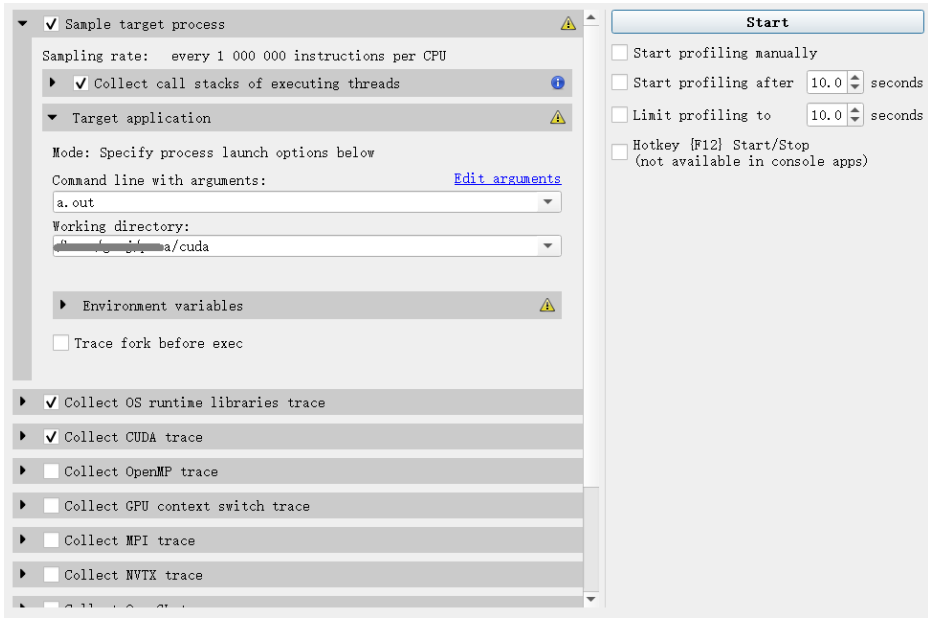
图 7.1: nsight

7.3 分析程序

如图，勾选要分析的部分，比如下面的 “Collect CUDA trace”.

然后在 Working directory 里面填上工作目录，在 Command line with arguments 里填上可执行文件和需要的参数。

在右边点击 start 即可开始分析。但是远程分析无法自动停止，因此需要估计好程序运行时间，然后点 stop。

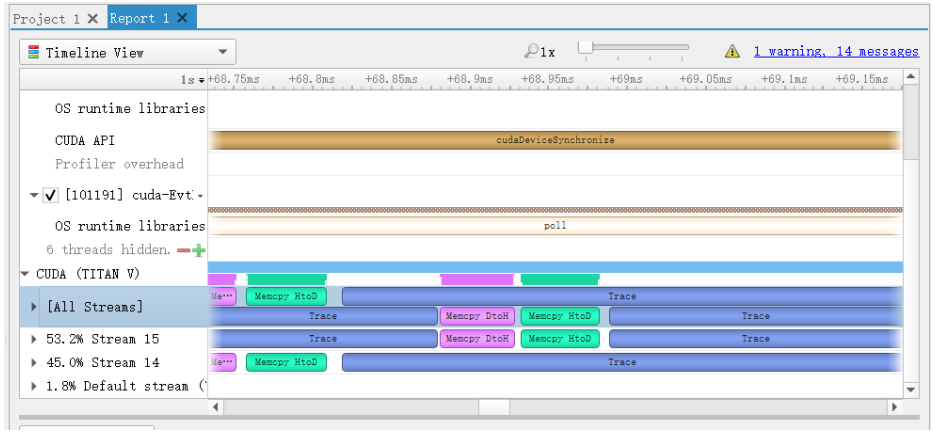


\begin{figure}
\caption{nsight_prof} \end{figure}

7.4 查看报告

停止以后，会出现一个 “Report 1” 标签页，里面有分析报告。上面可以看到时间轴，可以直观地看出各种调用占用的时间。

使用 Ctrl+ 鼠标滚轮可以缩放时间轴。



\begin{figure}
\caption{nsight_rep} \end{figure}

附录 A　汇编语言简介

要讲汇编语言，就不得不提计算机是如何工作的。众所周知，计算机（硬件）是看不懂你写的程序的。需要用编译器或者解释器将程序变成计算机能看懂的二进制码才可以。

汇编语言其实是一种 “助记符”。汇编语句和二进制机器码是一一对应的。相比于二进制机器码，汇编语言多少是可以读懂的。

A.1　基本操作

CPU 会做些什么事呢？基本就是通过内存地址从内存里取几个字节到自己的寄存器里，然后对各个寄存器进行一番操作，什么加减乘除啊移位啊与或非啊什么的，然后还能把结果再存到内存里。

那么，条件判断呀调用函数呀是怎么实现的呢？条件判断是分为两步的，第一步做比较，然后一些特别的寄存器会根据比较结果设置为相应的值；第二步，根据一个或几个特别的寄存器里面的值是 0 或者 1 选择是否跳转到另一个位置。而跳转实际就是把指向当前指令的指针加或减一个合适的值，而这个指针的值其实也是保存在一个寄存器中的。函数调用呢，就是先把一些需要保护的寄存器内容暂存到内存里，然后在合适的寄存器或内存地址里设置好要传递的参数，并且把当前的指令指针保存到内存中，然后跳转到相应函数位置。函数执行完毕后再把返回值放到指定的寄存器或者内存位置，然后再把保存起来的指令指针恢复到寄存器里。关于哪些寄存器需要调用方保护，哪些需要被调用方保护，用哪些寄存器传递参数和返回值则取决于编程语言的调用约定。

汇编语言中的数值拷贝（从内存到寄存器，寄存器到寄存器，寄存器到内存，立即数

到寄存器等等）被称作 “mov”，虽然事实上并不是移动，“mov” 完后原来位置的信息并不会被消掉。

> 立即数指的是直接有一个数字。从立即数到寄存器的 “mov” 相当于把寄存器设置成一个指定的值。

A.2 神奇的 “装载有效地址”

有一条指令 `lea`，是 “load effective address” 的缩写。这条指令本来是用于计算内存地址的，通过一个基地址，一个偏移量和每个元素的大小来计算出需要的地址。但是卑鄙的人类竟然动起了歪心思，用这条指令来做代数运算。

举个例子，比如要计算一个整数乘以 5 再加上 3，汇编代码有可能是这样的

```
lea 0x3(%rdi,%rdi,4), %rax
```

然后 rax 寄存器中的值就等于 rdi 寄存器的值乘上 5 然后再加三。

真正用于装载地址的时候，括号里面第一个值相当于基地址，第二个是偏移量，第三个是每一个偏移对应的字节数 (只能是 1,2,4,8)，括号前面的数字是以 byte 位单位的常数偏移量。

所有编译器就发现了这东西完全可以用于装载地址以外的用途。上面的例子中，计算了 rdi+rdi*4+3，也就是我们想要的乘 5 加 3.

A.3 c 语言中的内联汇编

不同编译器里使用内联汇编的方法可能有微小的不同。这里是 gcc 的情况

gcc 中，内联汇编写在 `__asm__` 里面。如果在 `__asm__` 后加了 `__volatile__` 则表示要求编译器不要对这里面的汇编代码进行优化。

这里用前面第一部分的 AVX 汇编举例

```
__asm__ __volatile__(
            "movq %0, %%rax \n\t"
            "movq %1, %%rbx \n\t"
            "movq %2, %%rcx \n\t"
            "movq %3, %%rdx \n\t"
            "movq %4, %%r8  \n\t"
            "shr  $2, %%r8  \n\t"
            "movq $0, %%r9  \n\t"
            "jmp  .check_%= \n\t"
            ".loop_%=:          \n\t"
            "shl $2, %%r9   \n\t"
            "leaq (%%rax, %%r9, 8), %%r10  \n\t"
            "vmovupd (%%r10), %%ymm0       \n\t"
            "leaq (%%rbx, %%r9, 8), %%r10  \n\t"
            "vmovupd (%%r10), %%ymm1       \n\t"
            "leaq (%%rcx, %%r9, 8), %%r10  \n\t"
            "vmovupd (%%r10), %%ymm2       \n\t"
            "vmulpd %%ymm0, %%ymm1, %%ymm3 \n\t"
            "vaddpd %%ymm2, %%ymm3, %%ymm3 \n\t"
            "leaq (%%rdx, %%r9, 8), %%r10  \n\t"
            "vmovupd %%ymm3, (%%r10)       \n\t"
            "shr $2, %%r9                  \n\t"
            "add $1, %%r9                  \n\t"
            ".check_%=:                    \n\t"
            "cmpq %%r8, %%r9               \n\t"
            "jl .loop_%=                   \n\t"
            :
            :"m"(a), "m"(b), "m"(c), "m"(d), "m"(N)
            :"%rax", "%rbx", "%rcx", "%rdx", "%r8", "%r9", "%r10",
             "%ymm0", "%ymm1", "%ymm2", "%ymm3", "memory"
            );
```

首先，每行写一句只是为了看起来方便。在 c 语言中，一个长的字符串是可以换行的，分到每一行后分别加引号，中间没有逗号就会当作一个长的字符串。

每一句后面加\n\t，否则多条汇编语句会在结果中连成一行，无法被汇编器读懂。

汇编代码一行一句，不用分号

__asm__ 的括号里首先要写的就是我们要执行的汇编代码之后用冒号隔开三个部分，分别是要写入的变量，要读取的变量和发生变动的寄存器。

变量前面引号里面 “m” 表示内存内容，用 “r” 则表示这个变量应该被放入寄存器（由编译器指定一个寄存器）。要写入的寄存器变量要写成 “=r”。

后面寄存器列表里面，把汇编代码中用到的寄存器都新进来就对了。似乎改变了内存的话要写 “memory”，但没有验证过，反正写了不会出问题。

在汇编代码里引用变量要用 “%”，百分号后面的数字表示变量的编号（后面从写入变量到读取变量从 0 开始连续编号，数一数是第几个）。由于这里使用了百分号，使用寄存器就要多写一个百分号，比如 “%eax” 就要变成 “%%eax”.

在寄存器外面加一个括号表示把这个寄存器中的值当作一个地址，要的是这个地址的内存里面的东西（除了装载有效地址的时候这个只是当作一个地址，不会实际访问内存）。

汇编代码中以冒号结尾的是一个标签，跳转的时候可以跳转到标签。最后编译好多二进制里是没有这个标签的，跳转会全部被翻译为跳转到相应的地址。在 gcc 的内联汇编里面，跳转标签要在后面加上 _%。

在 cmpq %%r8, %%r9 这句，比较了 r9 寄存器和 r8 寄存器中的值，并相应地设置了特别的用于比较和跳转的寄存器。后面紧跟的一句 jl .loop_%，是说如果判出来大于，就要跳转，否则不跳。

附录 B　浮点数的计算机表示

"The answer to the ultimate question of life, the universe and everything is 42."——The Hitchhiker's Guide to the Galaxy

这里只对双精度，单精度和半精度浮点数进行说明，不涉及 x87 的 80 位扩展精度浮点数。这几种浮点数都是按照 IEEE754 标准里规定的方式进行存储的。

计算机表示浮点数，其实用的是科学计数法，惊不惊喜，意不意外？只不过是二进制的科学计数法。计算机中的一个浮点数包含三个部分，符号、指数和尾数。下面用单精度浮点数来说明这三个部分是怎么组合成一个 32 位实体的。

B.1　单精度浮点数

一个单精度浮点数在内存中大概是下面这个样子的（不考虑实际存储按照大段序或小端序。左边是高位，右边是低位）。

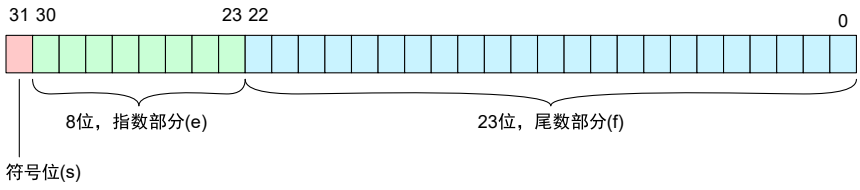


图 B.1: 单精度浮点数

所有的单精度浮点数分为三种，规格化的，非规格化的和特殊值。

B.1.1 规格化的单精度浮点数

大多数浮点数是规格化的。规格化的浮点数要求指数部分不为全零且不为全一。

符号位 (s) 表示浮点数的正负。0 表示正数，1 表示负数。

指数部分 (e) 表示一个有偏置的指数。非全零或全一的八位二进制可以表示从 1 到 254，这里偏置是-127。所以指数部分的范围是-126 到 +127.

尾数部分 (f) 之所以称为 “尾数”，是因为小数点前面的 1 并没有被表达在这一部分里面。根据科学计数法的定义，小数点前必须有且仅有一位，而且不是零。二进制中，这一位只能是 1. 于是我们通过这种方法相当于获得了额外的一位精度。

综上，规格化的浮点数表示的值可以写成

$$value = (-1)^s \times (1 + f \times 2^{-23}) \times 2^{e-127}$$

或者

$$value = (-1)^s \times 1.f_{22}f_{21} \cdots f_0 \times 2^{e-127}$$

B.1.2 非规格化的单精度浮点数

指数位全部为 0 的时候浮点数称为非规格化的浮点数。这部分浮点数提供了 0 的表示方法。

这部分浮点数就不适用科学计数法了，表示方法类似于无符号整数。

$$value = (-1)^s \times f \times 2^{-149}$$

或者

$$value = (-1)^s \times 0.f_{22}f_{21} \cdots f_0 \times 2^{-126}$$

指数固定为-126，这样就可以 “无缝” 地与规格化浮点数相连起来。当尾数部分全部为 0 地时候，这个浮点数就表示 0.

在标准中，+0 和-0 是两个数，它们只在一些微小的地方略有不同，一般不用在意。

B.1.3 特殊值

指数位全部是 1 的浮点数表示特殊值，包括 NAN (not a number) 和正负无穷大。当尾数全部为 0 时，根据符号位，表示正无穷或负无穷。尾数不全为 0 的则表示 NAN。当运算结果溢出的时候会得到正负无穷的结果，一些 “无意义” 的计算会导致 NAN。

B.2 双精度浮点数和半精度浮点数

表示方法和单精度浮点数一致，只不过指数部分有 11 位 (表示-1022 到 +1023)，尾数部分有 52 位。

半精度浮点数也是一样的，指数部分 5 位 (表示-14 到 +15)，尾数部分只有 10 位。

附录 C　高速缓存

在古代，是没有高速缓存的说法的，cpu 直接和内存通信。后来 cpu 越来越快，内存却跟不上 cpu 的速度了，于是人们设计了高速缓存。高速缓存离 cpu 更近，也就有了比内存更快的速度，然而高速缓存比较昂贵，因此空间非常有限。

现代的 cpu 一般有多级缓存，一级缓存比二级缓存快但比二级缓存小，二级缓存比三级缓存快但比三级缓存小，这样。这里按照最简单和原始的情况，也就是只有一级缓存的情况来描述缓存的性质。

对于程序来说，缓存是 “透明” 的，也就是说程序是感受不到缓存的存在的，所有的内存访问看起来都像是直接访问了内存。这种特性使得编程变得很容易，但是如果要针对缓存使用进行优化则变得比较困难。

C.1　局部性原理

局部性原理有两种形式：时间局部性和空间局部性。

时间局部性：一个程序具有良好的时间局部性的话，被引用过一次的存储器位置很有可能在**不久**的将来再次被引用。

空间局部性：一个程序具有良好的空间局部性的话，被引用过一次的存储器位置**附近**的位置很有可能在不久的将来被引用。

具有良好局部性的程序可以最大程度地利用缓存的特性。局部性原理和高速缓存好像有点像鸡和蛋的关系。

C.2 缓存的组态

缓存是分成许多**组**的，每一组又包含一些**行**，每一个行有一个由若干字节组成的块。每个行里还包含一个标记。

假设我们的缓存一共有 $S = 2^s$ 组（组的数目一定是 2 的某次方），每个组有 E 行，每一行包含 $B = 2^b$ 字节（B 也一定是 2 的某次方）。然后设计算机的所能给出的地址共 M 位（比如在 32 位系统里 $M = 32$），一条地址就按照下图分成三部分

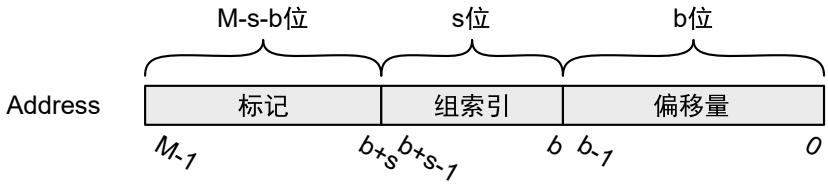


图 C.1: 内存地址和缓存的对应

每次访问一个地址的时候，cpu 会根据地址的组索引部分去找缓存中相应的组，然后讲标记部分和组里每一行的标记做对比，如果一致那么将直接读取缓存中的数据，避免访问内存。如果这一组所有的行的标记都与地址的标记不同，将会从这一组中消去一行，然后把目标内存缓存进来。（每次会缓存 2^b 字节。也就是这一组标志和索引对应的所有内存内容）。

注意到偏移量是 b，而缓存的每一行中有 2^b 个字节，cpu 会根据这个偏移量去缓存中找对应的字节。

关于缓存不命中的时候到底要消去缓存组中的哪一行，有各种不同的设计，比如随机消去一行，或者把最后一次使用时间最远的消去，或者把访问量最小的一行消去等等。

C.3 缓存优化

举个例子，在操作比较大的矩阵的时候，假设矩阵是按照行存储的（$address = col + row \times max\_col$），那么按照先行后列的方式访问可能会获得更快的速度。

在某些情况下，如果更加精细地设计一些，也许可以在每一行后面增加一些无意义的字节，来提高缓存命中的概率（比如需要交替访问多行，使得访问后面的行的时候不要消除前面行的缓存内容）。

C.4 如何查看 cpu 的缓存组态

- 查看 L1 缓存有多少组：

```
cat /sys/devices/system/cpu/cpu0/cache/index0/number_of_sets
```

上面 cpu0 表示第一个 cpu，服务器上/sys/devices/system/cpu/下会有 96 个 cpu。不要担心，它们的缓存参数是一致的。index0 表示该 cpu 里面的第一个缓存。经实验，index0 和 index1 是一级缓存，分别是数据和指令缓存；index2 是二级缓存；index3 是三级缓存。二级和三级缓存都是部分数据/指令的。

- 查看每一组有多少行：

```
cat /sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity
```

- 查看每一行有多少字节：

```
cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
```

- 查看缓存的大小

```
cat /sys/devices/system/cpu/cpu0/cache/index0/size
```

其实也可以算出来。上面每一行字节数乘以每一组行数再乘以组数即可。

- 其他信息也都在同意目录下，可以自行探索一下。

C.5 感受一下缓存不命中

还是请回我们的 muladd 程序。据查，我们的 cpu 的一级数据缓存每一行有 64 个字节。已知每个 double 占 8 个字节，也就是说我们的一行缓存最多可以缓存 8 个双精度浮点数（如果它们在内存里是对齐到 8 字节的话（一般来说的确是这么对齐的））。这回我们每次跳跃地进行计算，每次加 8，来尝试使得缓存无法命中：

```
__attribute__ ((noinline))
void muladd(double* a, double* b, double* c, double* d,
            unsigned long long N){
    unsigned long long i,j;
    for(i = 0; i < 8; i++){
        for(j = i; j < N; j+=8){
            d[j] = a[j] * b[j] + c[j];
        }
    }
}
```

根据上一节的命令，我们查到 CPU 的一级缓存是 32kB，也就是说最多可以存储 4096 个 double，所有无论如何，这些数字是不可能全部缓存起来的。（这也是最开始为什么选择了 8192 作为数组维度）。

显然，这个函数也是一样对每组 a[i]，b[i]，c[i] 进行一次计算。编译并运行，可以发现原来耗时约 20s 的程序，现在需要 30s。

二级缓存和三级缓存相对空间要大一些，而且是数据和指令共用的，一般就不对它们进行优化了（吧？）。

参考文献

nVidia (2020). Cuda toolkit documentation v11.0.3.

Randal E. Bryant, D. R. O. (2010). 深入理解计算机系统. 机械工业出版社, 2nd edition. ISBN 978-7-111-32133-0.

Xie, Y. (2020a). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.20.

Xie, Y. (2020b). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.29.