برنامه‌نویسی GPU در Java

دسترسی به GPU از طریق جاوا، باعث افزایش توان و قدرت قابل‌توجهی در برنامه می‌شود. در این مقاله نحوه‌ی کار GPU و نحوه‌ی دسترسی به GPU از طریق جاوا را توضیح میدهم.

ظاهرا برنامه‌نویسی GPU، مانند یک دنیای مجزا از برنامه‌نویسی جاواست... این موضوع عجیب نیست، چرا که اکثر منابع موجود و کاربردی برای جاوا، برای GPUها قابل‌اجرا نیستند!

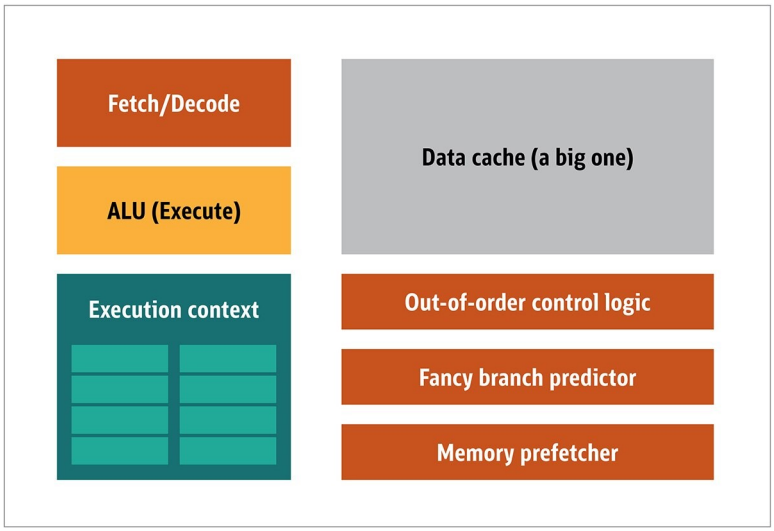
برای اینکه به موضوع اصلی برسیم، کمی در خصوص معماری و ساختار GPU، به همراه تاریخچه‌ی مختصری از آن را توضیح میدهم، که پرداختن به موضوع برنامه‌نویسی سخت‌افزار را آسان‌تر می‌کند. وقتی توضیح دادم که چگونه محاسبات GPU از محاسبات CPU متفاوت است، نشان خواهم داد که چگونه از GPUها در دنیای جاوا استفاده کنیم. در نهایت، فریم ورک و کتابخانه‌های معروف و دردسترس برای نوشتن کد جاوا و اجرای آن در GPUها را شرح خواهم داد و نمونه‌هایی را ارائه خواهم کرد.

فهرست/مندرجات

مقدمه

برای اولین بار در سال 1999، شرکت Nvidia، واحد پردازش گرافیکی (GPU) را به شهرت رساند. GPU یک پردازشگر خاص است که برای پردازش داده‌های گرافیکی، **پیش از انتقال به صفحه‌ی نمایش** طراحی شده است. در بیشتر موارد، GPU برخی محاسبات را از CPU ممکن می‌سازد که از CPU حذف و تخلیه شوند، در نتیجه درحالیکه حذف و تخلیه‌ی آنها را تسریع می‌کند، منابع CPU را نیز آزاد می‌کند. حاصل این کار این است که داده‌های ورودی بیشتری می‌توانند پردازش شوند و در رزولوشن خروجی بالاتری ارائه شوند، که در نتیجه، نمایش تصویری را جذاب تر و frame rate را تسریع می‌بخشد.

ماهیت پردازش 2D/3D اغلب به صورت دستکاری ماتریسی است، بنابراین می‌توان با یک رویکرد parallel، آن را کنترل و مدیریت کرد. یک رویکرد موثر برای پردازش تصویر چه خواهد بود؟ برای پاسخ به این سوال، بیایید معماری CPUهای استاندارد (شکل 1) را با GPUها مقایسه کنیم.

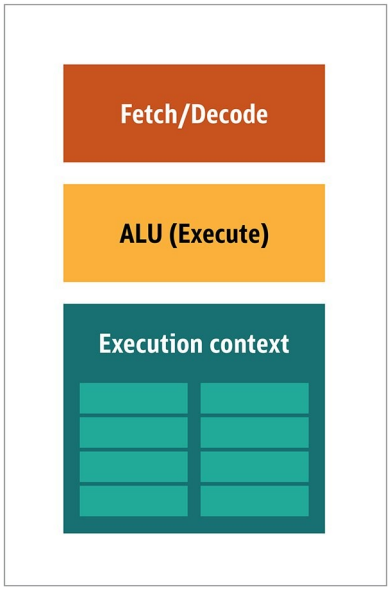


شکل1. معماری بلاک یک CPU

در CPU، عناصر پردازشگر واقعی، از جمله fetchers، واحد محاسبه و منطق(ALU) و مولفه های execution، تنها بخش کوچکی از کل سیستم هستند. برای سرعت بخشیدن به محاسبات نامنظم که با ترتیب غیرقابل پیش‌بینی به دست CPU می‌رسند، یک

حافظه‌ی cache بزرگ، سریع و گران قیمت، و همچنین انواع مختلف prefetcherها و پیش‌بینی‌کننده‌یbranch نیز وجود دارد.

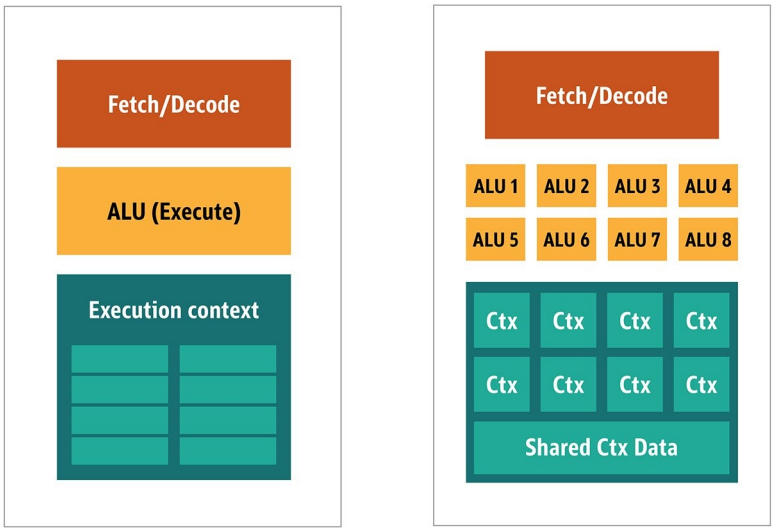
در GPU به تمامی این موارد نیازی ندارید، چرا که داده‌ها به شیوه‌ی قابل‌پیش‌بینی دریافت می‌شوند و GPU مجموعه عملیات محدودی را روی داده‌ها انجام می‌دهد. بنابراین، این امکان وجود دارد که یک پردازنده‌ی کوچک و ارزان قیمت را با معماری بلوک و مشابه با آنچه در شکل 2 نشان داده شده است، ایجاد کنیم.



شکل2. معماری بلوک برای یک هسته‌ی GPU ساده

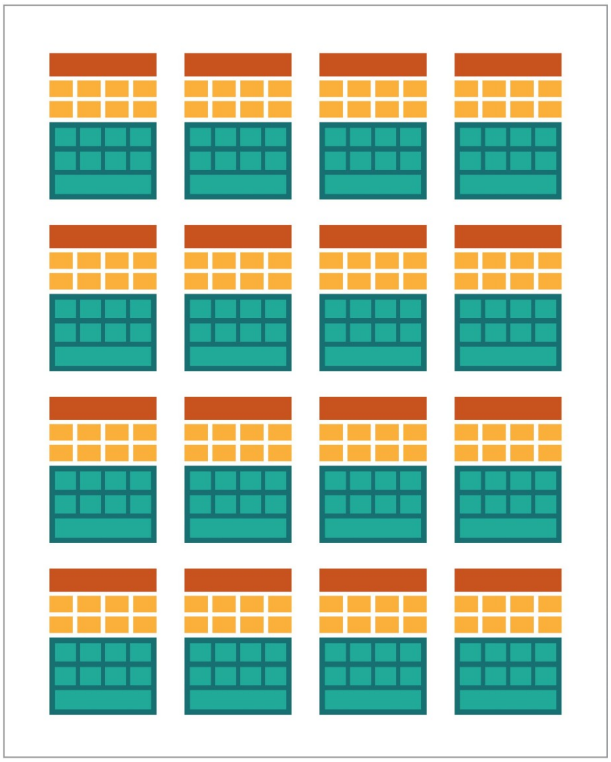
از آنجایی که این پردازنده‌ها ارزان قیمت هستند و داده‌ها را در تکه‌های parallel پردازش می‌کنند، قراردادن بسیاری از آنها برای عمل‌کردن به صورت parallel، آسان است. به این طراحی، «MIMD» یا چنددستوره، چندداده گفته می‌شود. (به صورت «میم،دی» تلفظ می‌شود.)

رروش دوم، یکsingle instruction، اغلب برای آیتم‌های چند داده به کار می‌رود. این روش با عنوان single instruction یا multiple data یا «SIMD» شناخته می‌شود. (به صورت «سیم،دی» تلفظ می‌شود.) در این طراحی، یک GPU واحد، دارای چندین ALU و زمینه‌ی اجرایی است و همچنین حاوی ناحیه‌ی کوچکی است که به shared context data اختصاص داده شده است. این طراحی در شکل 3 نشان داده شده است.



شکل3. مقایسه‌ی معماری بلوک GPU به سبک MIMD با طراحی SIMD.

ترکیب و ادغام پردازش SIMD و MIMD، حداکثر توان عملیاتی پردازش را فراهم می‌کند که در مورد آن به اختصار توضیح خواهم داد. در چنین طرحی، پردازنده‌های SIMD متعددی دارید که به صورت parallel در حال اجرا هستند، همانطور که در شکل 4 نشان داده شده است.



شکل4. اجرای چند پردازنده‌ی SIMD به صورت parallel که در اینجا، 16 هسته به همراه 128 مجموع ALU.

از آنجایی که تعدادی پردازنده‌ی کوچک و ساده دارید، می‌توانید آنها را برنامه‌ریزی کرده تا در خروجی، جلوه‌ها و صحنه‌های خاصی را به دست آورید.

اجرای برنامه‌ها بر روی GPU

بیشتر جلوه‌های visual اولیه در بازی‌ها، در واقع بر روی برنامه‌های کوچکی که روی GPU درحال اجرا هستند، hardcode می‌شدند و برای جریان داده‌ها از CPU به کار گرفته می‌شدند.

واضح است که حتی پیش از آن، این الگوریتم‌های تثبیت شده ناکافی بودند، مخصوصاً در طراحی بازی‌ها، که در آن نمایش visual یکی از اهداف اصلی فروش است. در واکنش به این موضوع، عرضه‌کنندگان یا فروشندگان بزرگ، دسترسی به GPUها را باز کردند و سپس برنامه‌نویسان و توسعه‌دهندگان نیز می‌توانستند برای آنها کدنویسی کنند.

روش معمول، نوشتن برنامه‌های کوچکی به نام shaders، به یک زبان مخصوص (معمولاً زیرمجموعه‌ای از زبان C) است و همچنین گردآوری آنها با یک کامپایلر خاص برای معماری مربوطه است. واژه‌ی Shaders به این دلیل انتخاب شده است که Shaders اغلب برای کنترل نورپردازی و جلوه‌های سایه‌زنی و هاشورزنی به کار می‌روند، اما هیچ دلیلی وجود ندارد که چرا آنها نمی‌توانند جلوه‌های ویژه‌ی دیگر را مدیریت و کنترل کنند...

هر تولیدکننده GPU، زبان برنامه‌نویسی و زیرساخت خاص خود را به منظور ایجاد shaders برای سخت‌افزار خود دارد. از این تلاش‌های به عمل آمده، چندین پلتفرم ایجاد شده اند. مهمترین آنها عبارتند از:

- **DirectCompute:** زبان shader اختصاصی/API از مایکروسافت که بخشی از Direct3D است و با DirectX10 آغاز می‌شود.
- **AMD FireStream:** تکنولوژی اختصاصی Radeon/ATI که توسط شرکت AMD متوقف شد.

- **OpenACC:** یک راهکار محاسبه‌ی parallel که با مشارکت چند عرضه‌کننده شکل گرفته است.
- **C++ AMP:** کتابخانه‌ی اختصاصی مایکروسافت برای paralleling داده‌ها در C++
- **CUDA:** پلتفرم اختصاصی Nvidia که از زیرمجموعه‌ی زبان C استفاده می‌کند.
- **OpenCL:** یک استاندارد مشترک که در اصل توسط شرکت Apple طراحی شده است، اما اکنون تحت مدیریت گروه Khronos است.

بیشتر اوقات، کارکردن با GPUها به معنای برنامه‌نویسی سطح پایین است و برای اینکه این مسئله برای توسعه‌دهندگان قابل درک باشد، چندین مفهوم abstract ارائه می‌شود. مشهورترین این مفاهیم، DirectX از شرکت مایکروسافت و OpenGL از گروه Khronos است. این APIها برای کدنویسی high-level هستند که می‌توانند به طور یکپارچه توسط توسعه‌دهنده، با GPU پیاده سازی شوند.

تا جاییکه می‌دانم، هیچ زیرساخت جاوایی وجود ندارد که از DirectX پشتیبانی کند، اما پیوند و وابستگی خوبی با OpenGL وجود دارد. در سال 2002، پیشنهاد JSR231 برای پرداختن به برنامه‌نویسی GPU داده شد، اما در سال 2008 از کار افتاد و تنها OpenGL 2.0 را پشتیبانی کرد. پشتیبانی از OpenGL در یک پروژه‌ی مستقل به نام JOCL ادامه پیدا کرد (که از OpenCL نیز پشتیبانی می‌کرد)، و در دسترس عموم قرار گرفت. به هر حال، بازی مشهور Minecraft با زیرساخت پروژه‌ی JOCL نوشته شد!

ظهور GPGPU

اگرچه جاوا و GPU باید به صورت یکپارچه و هماهنگ باشند، اما هنوز اینگونه نیستند. جاوا به شدت در شرکت‌ها، علوم داده، و بخش مالی مورد استفاده قرار می‌گیرد، در حالیکه محاسبات و قدرت پردازش بسیار زیادی لازم است. اینجاست که ایده‌ی «برنامه جامع محاسبه با واحد های پردازش گرافیکی» یا GPGPU مطرح می‌شود.

ایده‌ی استفاده از GPU در این روش، زمانی آغاز شد که تولیدکنندگان آداپتورهای گرافیک، شروع به بازکردن frame buffer کردند، که توسعه‌دهندگان را قادر به خواندن محتواها می‌کرد. برخی از هکرها هم به این مسئله پی بردند که می‌توانند از قدرت کامل GPU برای محاسبات جامع و کلی استفاده کنند. روش‌کار واضح و روشن بود:

1. کدگذاری یا رمزگذاری داده‌ها به عنوان آرایه‌ی bitmap
2. نوشتن یک shader برای پردازش آن
3. ارائه و ارسال هردوی آنها به کارت گرافیک
4. دریافت پاسخ از frame buffer
5. رمزگشایی داده‌ها از آرایه‌ی bitmap

توضیح این روش کار، ساده است. مطمئن نیستم که این فرآیند تا به حال به سختی در تولید مورد استفاده قرار گرفته باشد، اما این فرآیند عملی شده است.

سپس چندین محقق از دانشگاه استنفورد به دنبال راهی برای استفاده‌ی آسان‌تر از GPGPU بودند. در سال 2005، آنها BrookGPU را عرضه کردند که یک framework کوچک و شامل یک زبان، کامپایلر و runtime بود.

BrookGPU برنامه‌هایی را که در زبان برنامه‌نویسی stream Brook نوشته شده بودند را جمع‌آوری می‌کند، که نوع متفاوتی از ANSI C است. این راهکار می‌تواند ،OpenGL v1.3+ DirectX v9+، یا CTM شرکت AMD را برای محاسبات backend مورد هدف قرار دهد و هم بر روی ویندوز و هم بر روی گنولینوکس اجرا شود. نکته اینکه BrookGPU برای debugging، می‌تواند یک کارت گرافیک مجازی را بر روی CPU هم شبیه‌سازی کند.

در دنیای GPGPU، لازم است که داده‌ها را در دستگاه کپی کنید (دستگاه به GPU و بُردی که بر روی آن سوار می‌شود اشاره دارد)، منتظر GPU برای پردازش داده‌ها بمانید و سپس داده‌ها را دوباره به زمان‌اجرای اصلی کپی کنید. این امر باعث ایجاد تاخیر زیادی می‌شود.

در میانه‌ی دهه‌ی 2000، زمانی که این پروژه تحت پیشرفت و توسعه‌ی فعال بود، این میزان تاخیر تقریباً مانع استفاده‌ی گسترده از GPU ها برای محاسبات عادی می‌شد.

با این حال، بسیاری از شرکت‌ها، آینده را در این فناوری مشاهده کردند. چندین تولیدکننده کارت گرافیک، شروع به ارائه‌ی GPGPUها به همراه تکنولوژهای اختصاصی آنها کردند و دیگر تولیدکنندگان، گروه‌ها و ارتباطاتی را تشکیل دادند تا مدل‌های برنامه‌ریزی کلی‌تر و چندمنظوره ارائه کنند تا روی انواع بیشتری از دستگاه‌های سخت‌افزاری اجرا و پیاده‌سازی کنند.

اکنون، اجازه دهید دو فناوری بسیار موفق در محاسبات GPU، یعنی **OpenCL** و **CUDA** را بررسی کنیم و ببینیم که جاوا چگونه با آنها کار می‌کند.

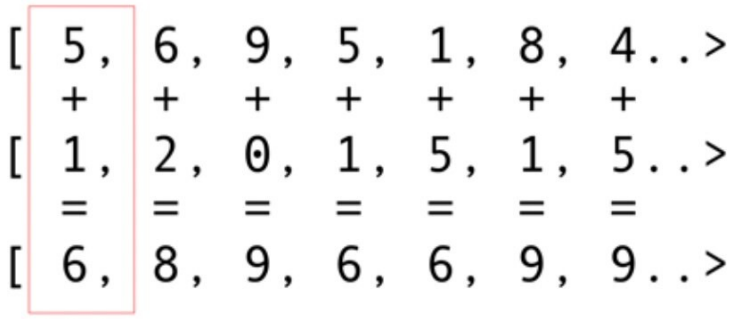
OpenCL و Java

همانند بسیاری از نرم افزارهای زیرساختی دیگر، OpenCL یک بستر پایه‌ای را در C فراهم کرده است. این امر به لحاظ فنی از طریق JNI یا JNA قابل دسترسی است، اما یک چنین دسترسی برای بیشتر توسعه‌دهندگان، کمی کار اضافی و مازاد را ایجاد خواهد کر.. اما خوشبختانه، این کار قبلاً توسط چندین کتابخانه صورت گرفته است: JogAmp ،JOCL، و JavaCL. ولی JavaCL پروژه‌ای است که از کار افتاده است! اما پروژه‌ی JOCL هنوز درحال کار کردن و تقریباً بروز است. در مثال‌های زیر از این پروژه استفاده خواهم کرد...

اما ابتدا، باید توضیح دهم که OpenCL چیست. همانطور که قبلاً اشاره کردم، OpenCL یک مدل بسیار کلی را ارائه می‌دهد که برای برنامه‌نویسی تمامی دستگاه‌ها، و نه تنها برای GPUها و CPUها، بلکه حتی برای پردازشگر DSP و FPGAs هم مناسب است.

بیایید ساده‌ترین نمونه، یعنی افزودن vector را بررسی کنیم که احتمالاً گویاترین و ساده‌ترین مثال است. شما دو آرایه‌ی عدد صحیح دارید که در حال افزودن و جمع آنها هستید و یک آرایه‌ی حاصل را دارید. شما یک عنصر از آرایه‌ی اول و یک عنصر از آرایه‌ی

دوم را انتخاب می‌کنید و سپس مجموع آنها را در آرایه‌ی حاصل قرار می‌دهید، همانطور که در شکل 5 نشان داده شده است.



شکل5. جمع محتوای دو آرایه و قرار دادن مجموع آنها در آرایه‌ی حاصل

همانطور که می‌بینید، عملیات جمع بسیار concurrent و parallelizable بوده و بنابراین قابل parallel شدن هستند. می‌توانید هریک از عملیات‌های جمع را در یک هسته‌ی GPU مجزا قرار دهید. یعنی اگر شما همانند کارت گرافیت 1080 شرکت Nvidia، چیزی برابر با 2048 هسته داشته باشید، می‌توانید 2048 عملیات جمع concurrent و parallel را انجام دهید! این به معنی آن است که پرفورمنس بالقوه‌ای از قدرت محاسباتی برای شما وجود دارند. در اینجا کد آرایه‌هایی با 10 میلیون عدد صحیح وجود دارد که از سایت JOCL گرفته شده است:

```
public class ArrayGPU
{
    /**
     * The source code of the OpenCL program
     */
    private static String programSource =
        "__kernel void "+
        "sampleKernel(__global const float *a,"+
        "             __global const float *b,"+
        "             __global float *c)"+
        "{"+
        "    int gid = get_global_id(0);"+
```

```
    "    c[gid] = a[gid] + b[gid];"+
    "}";

public static void main(String args[])
{
    int n = 10_000_000;
    float srcArrayA[] = new float[n];
    float srcArrayB[] = new float[n];
    float dstArray[] = new float[n];

    for (int i=0; i<n; i++)
    {
        srcArrayA[i] = i;
        srcArrayB[i] = i;
    }

    Pointer srcA = Pointer.to(srcArrayA);
    Pointer srcB = Pointer.to(srcArrayB);
    Pointer dst = Pointer.to(dstArray);
    // The platform, device type and device number
    // that will be used
    final int platformIndex = 0;
    final long deviceType = CL.CL_DEVICE_TYPE_ALL;
    final int deviceIndex = 0;
    // Enable exceptions and subsequently omit error checks in this sample
    CL.setExceptionsEnabled(true);
    // Obtain the number of platforms
    int numPlatformsArray[] = new int[1];
    CL.clGetPlatformIDs(0, null, numPlatformsArray);
    int numPlatforms = numPlatformsArray[0];
```

```
// Obtain a platform ID
cl_platform_id platforms[] = new cl_platform_id[numPlatforms];
CL.clGetPlatformIDs(platforms.length, platforms, null);
cl_platform_id platform = platforms[platformIndex];
// Initialize the context properties
cl_context_properties contextProperties = new cl_context_properties();
contextProperties.addProperty(CL.CL_CONTEXT_PLATFORM, platform);
// Obtain the number of devices for the platform
int numDevicesArray[] = new int[1];
CL.clGetDeviceIDs(platform, deviceType, 0, null, numDevicesArray);
int numDevices = numDevicesArray[0];
// Obtain a device ID
cl_device_id devices[] = new cl_device_id[numDevices];
CL.clGetDeviceIDs(platform, deviceType, numDevices, devices, null);
cl_device_id device = devices[deviceIndex];
// Create a context for the selected device
cl_context context = CL.clCreateContext(
    contextProperties, 1, new cl_device_id[]{device},
    null, null, null);
// Create a command-queue for the selected device
cl_command_queue commandQueue =
    CL.clCreateCommandQueue(context, device, 0, null);

// Allocate the memory objects for the input and output data
cl_mem memObjects[] = new cl_mem[3];
memObjects[0] = CL.clCreateBuffer(context,
    CL.CL_MEM_READ_ONLY | CL.CL_MEM_COPY_HOST_PTR,
    Sizeof.cl_float * n, srcA, null);
memObjects[1] = CL.clCreateBuffer(context,
    CL.CL_MEM_READ_ONLY | CL.CL_MEM_COPY_HOST_PTR,
```

```
    Sizeof.cl_float * n, srcB, null);
memObjects[2] = CL.clCreateBuffer(context,
    CL.CL_MEM_READ_WRITE,
    Sizeof.cl_float * n, null, null);
// Create the program from the source code
cl_program program = CL.clCreateProgramWithSource(context,
    1, new String[]{ programSource }, null, null);
// Build the program
CL.clBuildProgram(program, 0, null, null, null, null);
// Create the kernel
cl_kernel kernel = CL.clCreateKernel(program, "sampleKernel", null);
// Set the arguments for the kernel
CL.clSetKernelArg(kernel, 0,
    Sizeof.cl_mem, Pointer.to(memObjects[0]));
CL.clSetKernelArg(kernel, 1,
    Sizeof.cl_mem, Pointer.to(memObjects[1]));
CL.clSetKernelArg(kernel, 2,
    Sizeof.cl_mem, Pointer.to(memObjects[2]));
// Set the work-item dimensions
long global_work_size[] = new long[]{n};
long local_work_size[] = new long[]{1};
// Execute the kernel
CL.clEnqueueNDRangeKernel(commandQueue, kernel, 1, null,
    global_work_size, local_work_size, 0, null, null);
// Read the output data
CL.clEnqueueReadBuffer(commandQueue, memObjects[2], CL.CL_TRUE, 0,
    n * Sizeof.cl_float, dst, 0, null, null);
// Release kernel, program, and memory objects
CL.clReleaseMemObject(memObjects[0]);
CL.clReleaseMemObject(memObjects[1]);
```

```
        CL.clReleaseMemObject(memObjects[2]);
        CL.clReleaseKernel(kernel);
        CL.clReleaseProgram(program);
        CL.clReleaseCommandQueue(commandQueue);
        CL.clReleaseContext(context);
    }

    private static String getString(cl_device_id device, int paramName)
    {
        // Obtain the length of the string that will be queried
        long size[] = new long[1];
        CL.clGetDeviceInfo(device, paramName, 0, null, size);
        // Create a buffer of the appropriate size and fill it with the info
        byte buffer[] = new byte[(int)size[0]];
        CL.clGetDeviceInfo(device, paramName, buffer.length, Pointer.to(buffer),
null);
        // Create a string from the buffer (excluding the trailing \0 byte)
        return new String(buffer, 0, buffer.length-1);
    }
}
```

این کدنویسی اصلاً شبیه به جاوا نیست. این کد را بعداً توضیح خواهم داد، اکنون زمان زیادی را صرف آن نکنید، زیرا به طور مختصر، راهکار ساده‌تری را توضیح خواهم داد.

این کد به خوبی نوشته شده است، اما بیایید کمی روی آن بررسی کنیم. همانطور که می‌بینید، این کد بسیار شبیه به C است. این امر کاملاً طبیعی است، چرا که JOCL تنها به OpenCL اتصال دارد. در ابتدا، چند کد داخل یک رشته وجود دارند، و اینها در واقع مهمترین بخش هستند: این کد از سوی OpenCL جمع‌آوری و کامپایل شده‌است و سپس به کارت گرافیک فرستاده شده و در آنجا اجرا می‌شوند. این کد، Kernel نام دارد. آن را با

واژه‌ی Kernel سیستم عامل اشتباه نگیرید، این کد device است. (کدهای Kernel با زبان C و امثالش نوشته می‌شوند)

پس از وارد شدن Kernel، کد اتصال جاوا، device را تنظیم و سازماندهی کرده، داده‌ها را تقسیم‌بندی کرده و بافرهای حافظه ای مناسب را بر روی دستگاه ایجاد می‌کند که در آن داده‌ها و همچنین بافرهای حافظه، برای داده‌های result ذخیره‌سازی می‌شوند.

به طور خلاصه، «host code» وجود دارد که معمولاً binding زبان هم (در این مورد، زبان جاوا) دارد. علاوه بر این، «device code» نیز وجود دارد. شما همیشه آنچه روی host اجرا می‌شود و آنچه که باید روی device اجرا شود را مشخص کنید، زیرا host نیز device را کنترل می‌کند.

بیایید قابلیت‌ها و توانایی‌های SIMD را فراموش نکنیم. اگر سخت‌افزار شما از افزونه‌های SIMD پشتیبانی می‌کند، می‌توانید کدهای محاسباتی را بسیار سریع‌تر اجرا کنید. برای مثال، بیایید به کد Kernel ماتریس ضرب نگاهی بیندازیم. این کدی است که در raw string برنامه‌ی جاوا قرار دارد.

```
__kernel void MatrixMul_kernel_basic(int dim,
                  __global float *A,
                  __global float *B,
                  __global float *C){

    int iCol = get_global_id(0);
    int iRow = get_global_id(1);
    float result = 0.0;

    for(int i=0; i< dim; ++i)
    {
          result +=
          A[iRow*dim + i]*B[i*dim + iCol];
```

```
    }

    C[iRow*dim + iCol] = result;
}
```

از لحاظ فنی، این کد بر روی یک تکه یا قسمت از داده‌هایی کارساز خواهد بود که برای شما و توسط کتابخانه OpenCL به همراه دستورالعمل‌هایی که تهیه کرده‌اید، تنظیم شده است.

اگر کارت گرافیک شما از دستورالعمل‌های SIMD پشتیبانی می‌کند و می‌تواند vector هایی از چهار float را پردازش کند، بنابراین یک بهینه‌سازی کوچک روی کدها ممکن است کد قبلی را به کد زیر تبدیل کند:

```
#define VECTOR_SIZE 4
__kernel void MatrixMul_kernel_basic_vector4(
    size_t dim, // dimension is in single floats
    const float4 *A,
    const float4 *B,
    float4 *C)
{
    size_t globalIdx = get_global_id(0);
    size_t globalIdy = get_global_id(1);
    float4 resultVec = (float4){ 0, 0, 0, 0 };
    size_t dimVec = dim / 4;

    for(size_t i = 0; i < dimVec; ++i)
    {
        float4 Avector = A[dimVec * globalIdy + i];
        float4 Bvector[4];
        Bvector[0] = B[dimVec * (i * 4 + 0) + globalIdx];
```

```
        Bvector[1] = B[dimVec * (i * 4 + 1) + globalIdx];
        Bvector[2] = B[dimVec * (i * 4 + 2) + globalIdx];
        Bvector[3] = B[dimVec * (i * 4 + 3) + globalIdx];
        resultVec += Avector[0] * Bvector[0];
        resultVec += Avector[1] * Bvector[1];
        resultVec += Avector[2] * Bvector[2];
        resultVec += Avector[3] * Bvector[3];
    }

    C[dimVec * globalIdy + globalIdx] = resultVec;
}
```

با این کد، می‌توانید عملکرد را دوبرابر کنید...

اکنون شما GPU را برای دنیای جاوا باز کرده‌اید! اما به عنوان یک توسعه‌دهنده‌ی جاوا، آیا واقعاً می‌خواهید تمام این اتصال و پیوندها را انجام دهید؟ کد C بنویسید، و با چنین جزئیات سطح پایینی کار کنید؟ مطمئناً این کار را انجام نخواهیم داد. اما اکنون که کمی آگاهی در خصوص نحوه‌ی استفاده از معماری GPU را دارید، بیایید به راهکار دیگری فراتر از کد JOCL که آن را ارائه کردم، نگاهی بیندازیم.

CUDA و جاوا

CUDA همان راهکار شرکت Nvidia برای مشکلات و مسائل کدنویسی است. CUDA بسیاری از کتابخانه‌های آماده‌ی استفاده را برای عملیات‌های GPU، نظیر ماتریس‌ها، histogram ها و حتی شبکه‌های عصبی عمیق ارائه می‌دهد. این کتابخانه‌ی نوظهور شامل تعداد زیادی binding مفید است. در زیر مواردی که به پروژه‌ی JCuda مرتبط است را می‌بینید:

- JCublas: همه چیز در مورد ماتریس‌ها

- JCufft: انجام Fourier transform سریع
- JCurand: همه چیز در مورد اعداد تصادفی
- JCusparse: ماتریس‌های sparse
- JCusolver: فاکتورگیری
- JNvgraph: همه چیز در مورد گراف‌ها
- JCudpp: کتابخانه‌ی داده‌های paralleling و اولیه‌ی CUDA و برخی از الگوریتم‌های مرتب‌سازی
- JNpp: پردازش تصویر بر روی GPU
- JCudnn: کتابخانه‌ی شبکه‌ی عصبی

با استفاده از JCurand که اعداد تصادفی را تولید می‌کنم، توضیح خود را ادامه خواهم داد. می‌توانید آن را در کد جاوا استفاده کنید، بدون اینکه به دیگر زبان‌های Kernel خاص نیاز داشته باشید. برای مثال:

```
...
int n = 100;
curandGenerator generator = new curandGenerator();
float hostData[] = new float[n];
Pointer deviceData = new Pointer();
cudaMalloc(deviceData, n * Sizeof.FLOAT);
curandCreateGenerator(generator, CURAND_RNG_PSEUDO_DEFAULT);
curandSetPseudoRandomGeneratorSeed(generator, 1234);
curandGenerateUniform(generator, deviceData, n);
cudaMemcpy(Pointer.to(hostData), deviceData,
        n * Sizeof.FLOAT, cudaMemcpyDeviceToHost);
System.out.println(Arrays.toString(hostData));
curandDestroyGenerator(generator);
cudaFree(deviceData);
```

در اینجا برای ایجاد اعداد تصادفی با کیفیت بالا و براساس برخی عملیات‌های ریاضیاتی قوی، از GPU استفاده می‌شود.

در JCuda، می‌توانید کد CUDA بصورت کلی بنویسید و آن را با تعدادی فایل JAR به CLASSPATH خود، از جاوا بخواهید. برای مشاهده‌ی نمونه‌های بیشتر، نوشته‌ها و مستندات JCuda را ببینید.

ماندن در بالای کد low-level

همه‌ی این موارد عالی به نظر می‌رسد، اما تشریفات زیاد، تنظیم و آماده‌سازی بسیار زیاد، و بسیاری از زبان‌های مختلف برای اجرا و راه‌اندازی وجود دارند. آیا راهی وجود دارد که از GPU، حداقل به صورت جزئی استفاده کنیم؟

چه می‌شود اگر که شما دیگر نخواهید به تمامی این موارد CUDA، OpenCL و دیگر چیزهای internal فکر کنید؟ چه می‌شود اگر که شما بخواهید در جاوا کدنویسی کنید و در مورد دیگر چیزهای internal فکر نکنید؟ پروژه‌ی Aparapi می‌تواند به شما در این امر کمک کند. Aparapi مخفف واژه‌ی «a parallel API» است. من به این پروژه به عنوان نوعی خواب زمستانی برای برنامه‌نویسی GPU نگاه می‌کنم که از OpenCL در بطن آن استفاده می‌کند. بیایید به یک نمونه از ساخت vector، نگاهی بیندازیم.

```
public static void main(String[] _args)
{
    final int size = 512;
    final float[] a = new float[size];
    final float[] b = new float[size];

    /* fill the arrays with random values */
    for (int i = 0; i < size; i++)
          {
```

```
        a[i] = (float) (Math.random() * 100);
        b[i] = (float) (Math.random() * 100);
    }

    final float[] sum = new float[size];

    Kernel kernel = new Kernel()
    {
        @Override public void run()
        {
            int gid = getGlobalId();
            sum[gid] = a[gid] + b[gid];
        }
    };

    kernel.execute(Range.create(size));

    for(int i = 0; i < size; i++)
    {
        System.out.printf("%6.2f + %6.2f = %8.2f\n", a[i], b[i], sum[i])
    }

    kernel.dispose();
}
```

این نمونه، یک کد خالص جاوا است (که از مستندات Aparapi برگرفته شده است)، اگرچه در این کد می‌توانید برخی از اصطلاحات خاص دامنه‌ی GPU، نظیر "Kernel" و " getGlobalId" را مشاهده کنید. شما هنوز نیاز دارید که درک کنید، این پردازنده‌ها چگونه برنامه‌ریزی شده اند، اما می‌توانید به GPGPU به شیوه‌ای Java-Friendly نزدیک شوید.

علاوه بر این، Aparapi یک راه و روش آسان برای bind زمینه‌های OpenGL به لایه‌های زیرین OpenCL فراهم می‌کند، و از این رو داده‌ها را قادر می‌سازد تا کاملاً بر روی کارت گرافیک باقی بمانند و در نتیجه، از مشکلات memory latency جلوگیری کنند.

اگر لازم است محاسبات مستقل زیادی انجام شود، بنابراین پروژه‌ی Aparapi را مدنظر قرار دهید. این مجموعه‌ی غنی از نمونه‌ها، برخی موارد استفاده را برای شما فراهم می‌کنند که برای انجام محاسبات paralleling حجیم و گسترده، کامل و بی‌نقص هستند.

علاوه بر این، چندین پروژه‌ی دیگر مانند TornadoVM وجود دارد که به صورت خودکار محاسبات مناسب را از CPU به GPU انتقال می‌دهند و در نتیجه، بهینه‌سازی گسترده ای را ممکن می‌سازد.

نتیجه‌گیری

اگرچه برنامه‌های کاربردی زیادی وجود دارند که در آنها GPUها می‌توانند مزایایی را به همراه داشته باشند، اما ممکن است بگویید که هنوز چند مانع بر سر راه وجود دارد... اما با این حال، جاوا و GPUها می‌توانند به همراه یکدیگر کارهای بزرگی را انجام دهند. در این مقاله، تنها به **بررسی سطحی** این موضوع گسترده پرداختم. هدف من نشان دادن گزینه‌های سطح بالا و سطح پایین برای دسترسی به GPU از جاوا بود. بررسی و پژوهش در این حوزه، مزایای عملکردی زیادی را به همراه دارد، مخصوصاً برای مشکلات پیچیده‌ای که نیازمند محاسبات متعددی هستند که می‌توانند به صورت parallel انجام گیرند.

مترجم: یوشا آل ایوب

منبع: https://blogs.oracle.com/javamagazine/programming-the-gpu-in-java