基于任务的异步并行运行时系统
--HPX库介绍

Task-based parallel asynchronous runtime system

李健

2022. 08. 18

提纲

- ParallelX model的研究背景
- 异步运行时系统(AMT)
- HPX的架构及方法
- HPX的使用举例
- HPX在ARM机器上的测试

华罗庚烧水泡茶的故事，烧水泡茶的办法有三种：

办法甲：先做好准备工作，洗开水壶、茶壶、茶杯，拿茶叶。一切就绪后，灌水，烧水，等水开了泡茶喝。 20'

办法乙：洗净开水壶后，灌水，烧水。等水开了之后，洗茶壶、茶杯，拿茶叶，泡茶喝。 20'

办法丙：洗净开水壶后，灌水，烧水。利用等待水开的时候，洗茶壶、茶杯，拿茶叶，等水开了泡茶喝。 16'

如果是你，你会选哪种方法呢？

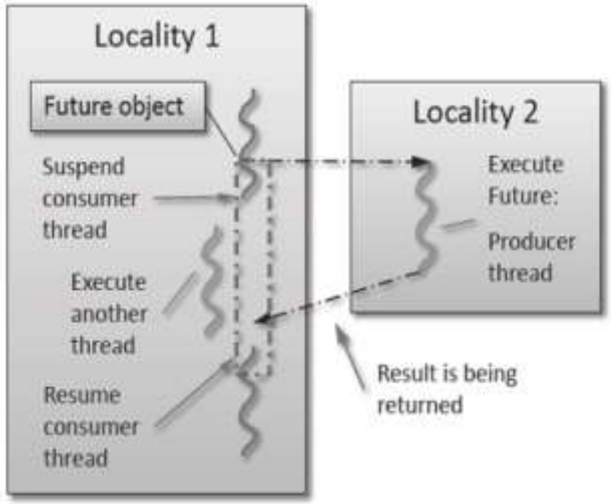


Fig. 2.1: Schematic of a future execution.

AMD ZEN4 EPYC(96核, 192线程) L3 cache 32MB

Intel Gen13 Raptor Lake L2+L3 Cache 68MB

鲲鹏920处理器片上系统采用三级Cache结构，每个处理器内核集成64KB的L1 I Cache(L1指令Cache)和64KB的L1 D Cache(L1数据Cache)，每核独享512KB L2 Cache，处理器还配置了L3 Cache，平均每核容量为1MB。

片内Cache： 两级片内Cache，包括固定大小为64KB的L1 I Cache和64KB的L1 D Cache，再加上每个处理器内核私有的8路512KB L2 Cache。L1 Cache和L2 Cache行大小均为64B(字节)。

一、ParallelX model的研究背景

新的并行化方法应强调以下特性：

◆Scalability – strongly scale to Exascale levels of parallelism;

◆Programmability –reduce the burden on high performance programmers;

◆Performance Portability– eliminate or significantly minimize requirements for porting to future platforms;

◆Resilience – properly manage fault detection and recovery at all components of the software stack;

◆Energy Efficiency – maximally exploit dynamic energy saving opportunities, leveraging the tradeoffs between energy efficiency, resilience, and performance.

一、ParallelX model的研究背景

FLOPS: floating point operations per second

理论峰值和实际峰值

2种形式的线性度(scalability)：

◆Strong Scaling: Amdahl’s Law

◆Weak Scaling: Gustafson’s Law

一、ParallelX model的研究背景

超大规模HPC的瓶颈(SLOW):

- ◆Starvation occurs when there is insufficient concurrent work available to maintain high utilization of all resources.
- ◆Latencies are imposed by the time-distance delay intrinsic to accessing remote resources and services.
- ◆Overhead is work required for the management of parallel actions and resources on the critical execution path, which is not necessary in a sequential variant.
- ◆Waiting for contention resolution is the delay due to the lack of availability of oversubscribed shared resources.

二、异步运行时系统（Asynchronous many-task systems）

典型的MPI+OpenMPI模型中的reduction barrier的执行模式：

使用fork-join模式的严格顺序执行的局部规约阶段，然后就是诸如集合操作的全局同步原语。

```
compute_region() {
  while (some_condition()){
    #pragma omp parallel
    {
       //execute shared memory parallel region
    }
    //global reduction
    MPI_Allreduce()
  }
}
```

Listing 1: **General MPI+OpenMP pattern for a two-phase reduction**

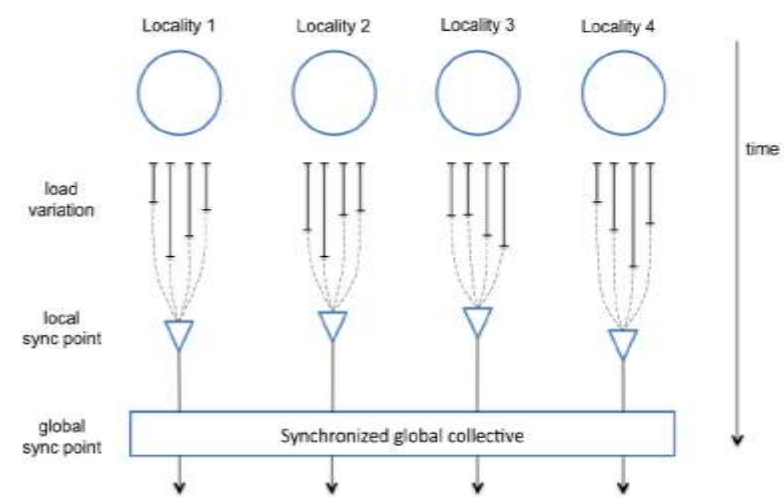
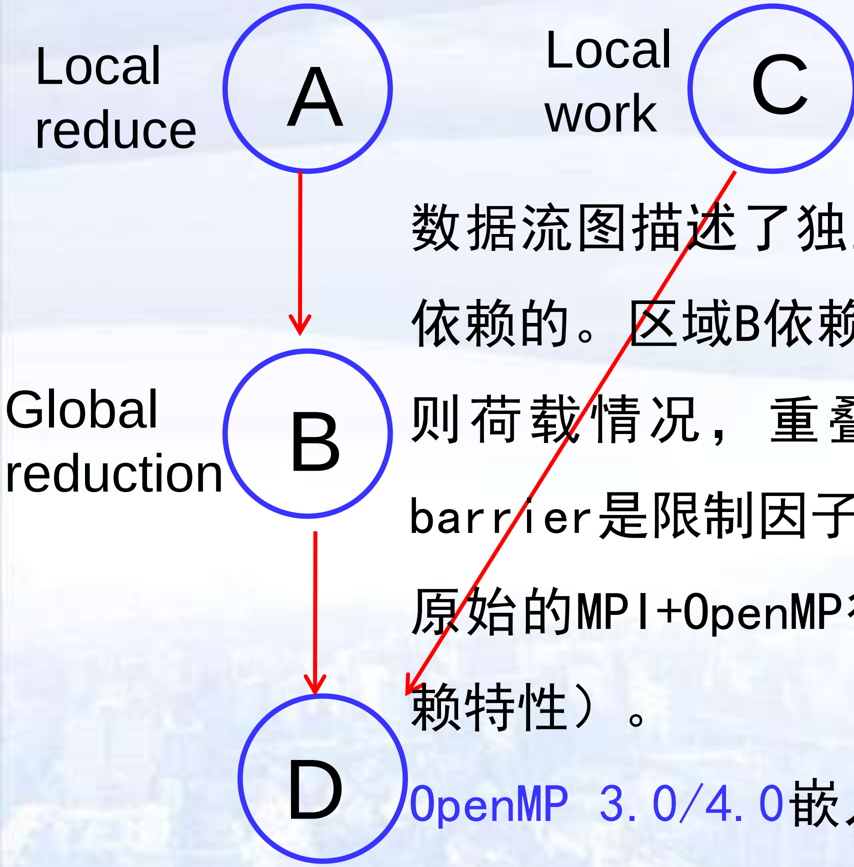


Fig. 7: **Benchmark overview for computation load injection**

二、异步运行时系统（Asynchronous many-task systems）

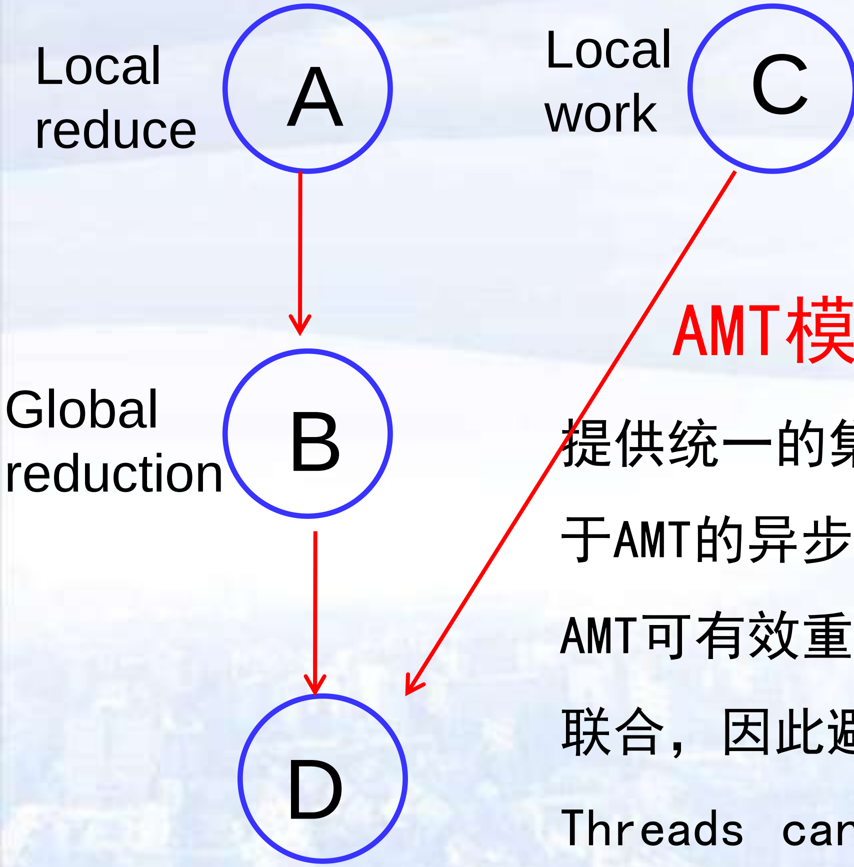


数据流图描述了独立区域A和C（没有直接联系），区域B，D是依赖的。区域B依赖区域A，然后区域D依赖区域B和C。对于不规则荷载情况，重叠区域C和B非常有益。但是，隐式的同步barrier是限制因子，不能隐藏区域A中的不规则性。

原始的MPI+OpenMP很难完全利用计算资源（具有并行的数据依赖特性）。

OpenMP 3.0/4.0嵌入动态循环调度和任务并行化技术，例如编程结构#pragma omp task，#pragma omp sections和nested regions，但这增加软件复杂度，且性能调优困难。

二、异步运行时系统(Asynchronous many-task systems)



AMT模式：

提供统一的集合方法，甚至是在不均衡荷载情况，这得益于AMT的异步设计。

AMT可有效重叠域（A，B）的计算与通信，将他们与区域C联合，因此避免同步的等待时间，增加输出。

Threads can compensate for late comers by taking up more work while waiting for a collective communication operation to complete.

二、异步运行时系统(Asynchronous many-task systems)

State-of-art: 基于任务(task-based)的并行模型分类：

- ◆并行编程库：TBB, PPL, Qthreads
- ◆语言拓展：Intel Cilk Plus, OpenMP 3.0/4.0
- ◆试验性的编程语言：Chapel, Charm++(1993), X10

上述方法，一些使用基于futures编程模型，有一些使用基于dataflow处理依赖的任务并行。大多数基于任务的编程模型处理单节点层级并行。 OpenMP 3.0/4.0是唯一支持FORTRAN语言的task-based编程模型，OpenMP 4.0引入task dependencies，着眼于fork-join编程。 Intel TBB是C++语言，提供基于任务的codelet风格的执行。

上述方法都缺乏统一的API和处理分布式并行计算的解决方案。

二、异步运行时系统（Asynchronous many-task systems）

HPX的优势：

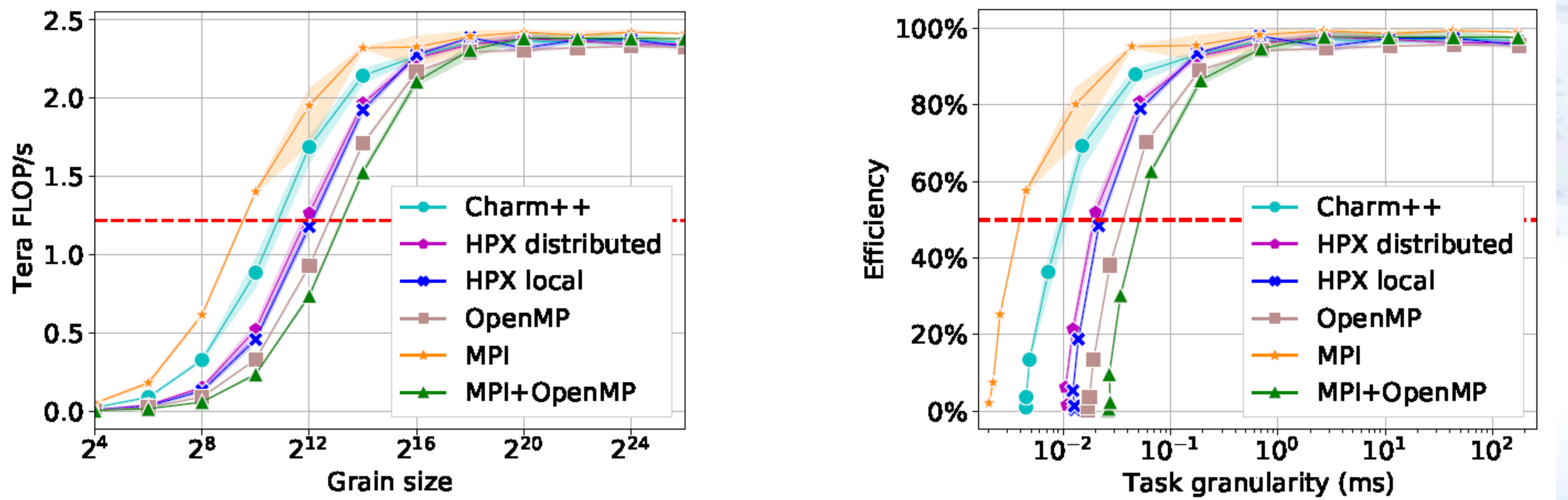
◆HPX编程API符合C++11/14标准；

◆HPX统一执行远程和局部操作；

◆HPX不使用新的语法和语义，HPX实施C++11的语法和语义，提供统一的API依赖于广泛接受的编程接口。因此，这有利于迁移legacy code；

◆HPX可使用MPI作为迁移的通信平台，同时HPX还可作为OpenMP和CUDA的后端，便于迁移遗留代码。

二、异步运行时系统（Asynchronous many-task systems）



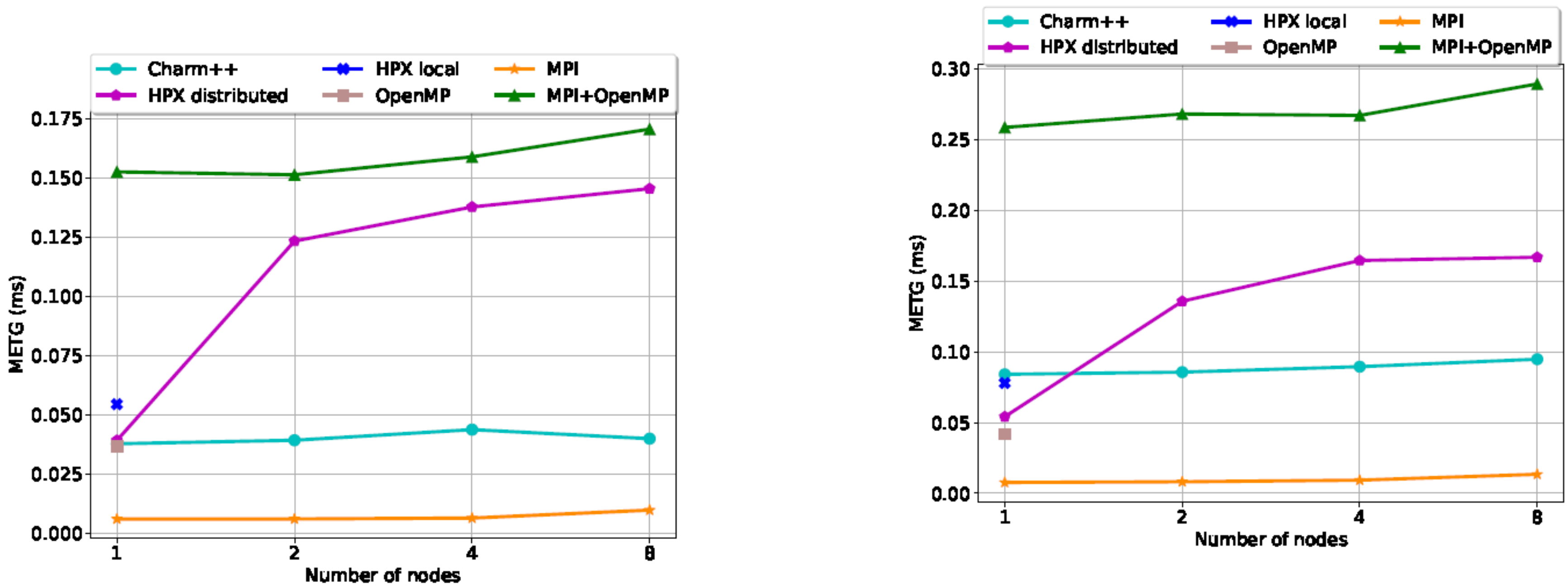
(a) Tera FLOP/s vs grain size.

(b) Efficiency vs task granularity

Figure 1: Stencil pattern, 1 node (48 cores), 48 tasks.

Task granularity = wall time×number of cores/number of tasks

Wu Nanmiao et al. 2022. Quantifying overheads in charm++ and hpx using task bench. arXiv:2207.12127v1 [cs.DC] 21 Jul 2022



(a) Stencil pattern, overdecomposition 8 (8 tasks per core).

(b) Stencil pattern, overdecomposition 16 (16 tasks per core).

Figure 2: METG of each system with varying number of nodes for different overdecomposition. METG is short for Minimum Effective Task Granularity, is an efficiency-constrained metric for runtime-limited performance, introduced in Task Bench paper [8].

多节点的分布式并行： METG (Minimum Effective Task Granularity)

METG越小越好，越平越好

二、异步运行时系统(Asynchronous many-task systems)

AMT发展的一些问题：

1、overheads，并行化的暴露，荷载均衡，调度策略，编程接口

2、整合众核硬件系统，避免内在的数据转移延迟

3、在AMR情况下，一方面，荷载均衡策略和随时间变化的数据结构，需要开发者控制；另一方面，显示出很大的不可预测行为，数据结构需要自适应控制

4、AMT需要硬件架构支持，硬件最小化overhead、有限延迟和暴露更多的并行性和扩展性

Thomas Sterling, Matthew Anderson, Maciej Brodowicz. 2017. A Survey: Runtime Software Systems for High Performance Computing

三、HPX的架构及方法

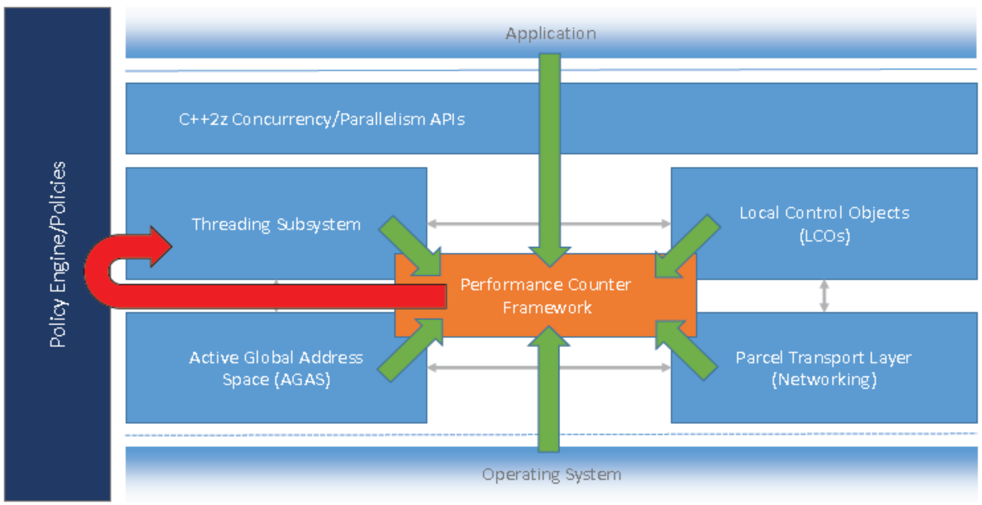


Figure 1: Sketch of HPX's architecture with all the components and their interactions.

Kaiser et al., (2020)

Threading Subsystem：轻量级线程管理器，降低不同线程协调执行时的同步成本。

Active Global Address Space (AGAS)：通过对象迁移，实现荷载均衡。

Parcel Transport Layer：active-message networking layer。默认HPX支持TCP/IP, MPI和libfabric。

Performance counters：实施现场性能监控。

Policy Engine/Policies

Accelerator Support (CUDA/openCL)

Local Control Objects：HPX提供hpx::async and hpx::future等，轻量级线程替代global barrier。

C++ Standards conforming API

Kaiser et al., (2020). HPX - The C++ Standard Library for Parallelism and Concurrency. Journal of Open Source Software, 5(53): 2352.

三、HPX的架构及方法

work items; work groups

threads; warp

并行化执行：

（1）execution restrictions：确保应用于work items的线程安全（即可以并行运行，或者必须串行运行）；

（2）work items必须在哪个sequence中执行（即该work items依赖于结果的获取性）；

（3）work items在何处执行（即"on this core"，"on the node"，"on this NUMA domain" or "wherever this data item is located"等）；

（4）在相同执行线程上的运行任务的粒度大小控制（即'执行的各线程应该运行相同数目的work items'）。

Hartmut Kaiser et al. Higher-level Parallelization for Local and Distributed Asynchronous Task-Based Programming. 2015

三、HPX的架构及方法

上述属性，HPX有对应的定义，见下表。HPX方法（实施一系列操作）对应相同syntax and semantics的C++类型。

Property	C++ concept name
Execution *restrictions*	execution_policy
Sequence of execution	executor
Where execution happens	executor
Grain size of work items	executor_parameter

Kaiser et al. 2015

ParallelX model不是一个新鲜的概念，C++并发编程规范早就有了。

N4501: Working Draft: Technical Specification for C++ Extensions for Concurrency. Technical report, , 2015. http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2015/n4501.html.

三、HPX的架构及方法

Property	C++ concept name
Execution *restrictions*	execution_policy
Sequence of execution	executor
Where execution happens	executor
Grain size of work items	executor_parameter

Execution Policies定义：

an object that expresses the requirements on the ordering of functions invoked as a consequence of the invocation of a standard algorithm

Parallelism TS定义了三种execution_policy， HPX中增加了par(task)和seq(task)这两个task执行策略，任务执行策略植入算法后立即返回，返回到future对象的激活位置，表征算法的最终结果。

Policy	Description	Implemented by
seq	sequential execution	Parallelism TS, HPX
par	parallel execution	Parallelism TS, HPX
par_vec	parallel and vectorized execution	Parallelism TS
seq(task)	sequential and asynchronous execution	HPX
par(task)	parallel and asynchronous execution	HPX

par_simd (2021)

Table 2: The execution policies defined by the Parallelism TS and implemented in HPX (HPX does not implement the par_vec execution policy as this requires compiler support).

Kaiser et al. 2015

三、HPX的架构及方法

Property	C++ concept name
Execution *restrictions*	execution_policy
Sequence of execution	executor
Where execution happens	executor
Grain size of work items	executor_parameter

Kaiser et al. 2015

Executor定义："an object responsible for creating execution agents on which work is performed, thus abstracting the (potentially platform-specific) mechanisms for launching work".

HPX依靠executor_traits利用executors，一个executor实施async_execute，返回一个future 对象，表示一个异步函数激活的结果。从这个实例，同步的execute可以由executor_traits合成，或者实施，提交给executor；

HPX将不同的预定义的executor类型暴露给用户。除了一般的串行和并行executor，用作默认为seq和par执行策略，HPX实施executor封装特殊的schedulers。

三、HPX的架构及方法

Property	C++ concept name
Execution *restrictions*	execution_policy
Sequence of execution	executor
Where execution happens	executor
Grain size of work items	executor_parameter

Kaiser et al. 2015

HPX中增加了executor_parameters的概念，允许控制工作的粒度，即：相同的执行线程上执行哪个以及多少work items。这与OpenMP static或guided调度原语很像，但OpenMP调度原语不是C++类型，HPX属于executor_parameters的类型可以在运行时做决策。

在运行时决策的情况，executor_parameters还允许定义某并行操作可以使用多少处理单元（核心）。

用户可以改造executor_parameters，适应特殊的应用。

四、HPX的使用举例

1）使用C++17标准定义的execution policies，实施HPX的并行算法API。并行算法通过增加一个形参（称为execution policy），拓展经典的STL算法。hpx::execution::seq执行串行计算；hpx::execution::par执行并行计算。

```
#include <hpx/hpx.hpp>
#include <iostream>
#include <vector>

int main()
{
 std::vector<int> values = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// Compute the sum in a sequential fashion
int sum1 = hpx::reduce(
hpx::execution::seq, values.begin(), values.end(), 0);
std::cout << sum1 << '\n'; // will print 55

// Compute the sum in a parallel fashion based on a range of values
int sum2 = hpx::ranges::reduce(hpx::execution::par, values, 0);
std::cout << sum2 << '\n'; // will print 55 as well

return 0;
}
```

Kaiser et al., (2020)

四、HPX的使用举例

2）计算sin(x)的泰勒级数： $\sin(x) \approx= \sum_{n=0}^{N} (-1)^{n-1} \frac{x^{2n}}{(2n)!}$ 。区间[0, N]分成2部分[0, N/2]和[N/2+1, N]，使用hpx::async异步计算。

```
#include <hpx/hpx.hpp>
#include <cmath>
#include <iostream>

// Define the partial taylor function
double taylor(size_t begin, size_t end, size_t n, double x)
{
 double denom = factorial(2 * n);
 double res = 0;
 for (size_t i = begin; i != end; ++i)
 {
 res += std::pow(-1, i - 1) * std::pow(x, 2 * n) / denom;
 }
 return res;
}

int main()
{
 // Compute the Talor series sin(2.0) for 100 iterations
 size_t n = 100;

 // Launch two concurrent computations of each partial result
 hpx::future<double> f1 = hpx::async(taylor, 0, n / 2, n, 2.);
 hpx::future<double> f2 = hpx::async(taylor, (n / 2) + 1, n, n, 2.);

 // Introduce a barrier to gather the results
 double res = f1.get() + f2.get();
 // Print the result
 std::cout << "Sin(2.) = " << res << std::endl;
}
```

Kaiser et al., (2020)

四、HPX的使用举例

HPX库自带的示例源码

1d_stencil

Matrix Transposition

Fibonacci

……

```
int fib(int x)
{
    if (n == 1 || n == 2)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Benchmark

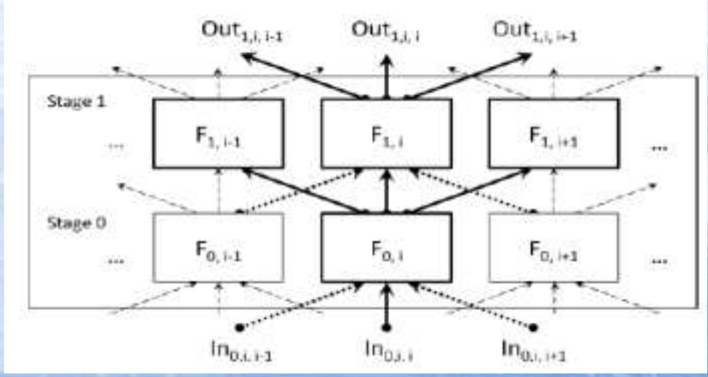
Mini-apps(mini-Ghost & HPCG)

STREAM

Octopus: an HPX octree-based 3D AMR framework

nast_hpx

DGSWE_v2



四、HPX的使用举例

3）OP2库的airfoil mini-app测试源码改造：

一共有5个循环，包含直接循环和间接循环。

```
op_par_loop_adt_calc("adt_calc",cells,
op_arg_dat(data_b0,...),
...,
op_arg_dat(data_bn,...));
```

四、HPX的使用举例

```
// 基于OpenMP 2.x并行模式，见 openmp/adt_calc_kernel.cpp
#pragma omp parallel for
for(int blockIdx=0; blockIdx<nblocks; blockIdx++) {
   int blockId =    //based on the blockIdx in OP2 API
   int nelem =      //based on the blockId
   int offset_b =   //based on the blockId
   for ( int n=offset_b; n<offset_b+nelem; n++ ) {
          .
          .
          .
       adt_calc(...);  // 用户根据OP2 API写好的核函数
   }
}
```

四、HPX的使用举例

```
void op_par_loop_adt_calc(char const *name,
              op_set set, op_arg arg0, op_arg arg1)
{
              .
static_chunk_size scs(SIZE);            // executor parameters
auto r=boost::irange(0, nblocks);
hpx::parallel::for_each(par.with(scs),  // execution policy
    r.begin(),r.end(),[&](std::size_t blockIdx){

    int blockId =  //based on the blockIdx in OP2 API
    int nelem =    //based on the blockId
    int offset_b =  //based on the blockId

    for ( int n=offset_b; n<offset_b+nelem; n++ ){
                         .
          adt_calc(...);   // user's kernel
    }
});
```

五、HPX在ARM机器上的测试

HPX在3个ARM集群上做了测试，包括：ThunderX2 (Marvell, USA), Kunpeng 916 (Huawei, China)和A64FX (Fujitsu, Japan)

STREAM COPY

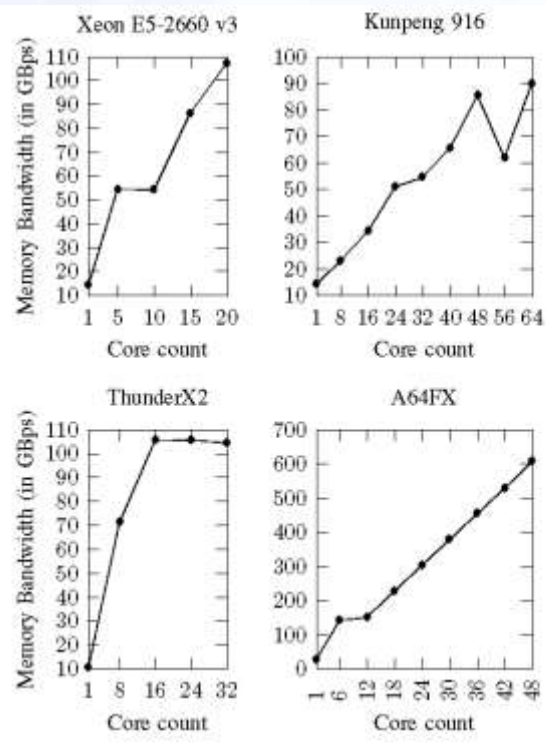


Fig. 2: Memory Bandwidth results using the STREAM COPY Benchmark with an array size of 128 million elements

五、HPX在ARM机器上的测试

1D Stencil算例

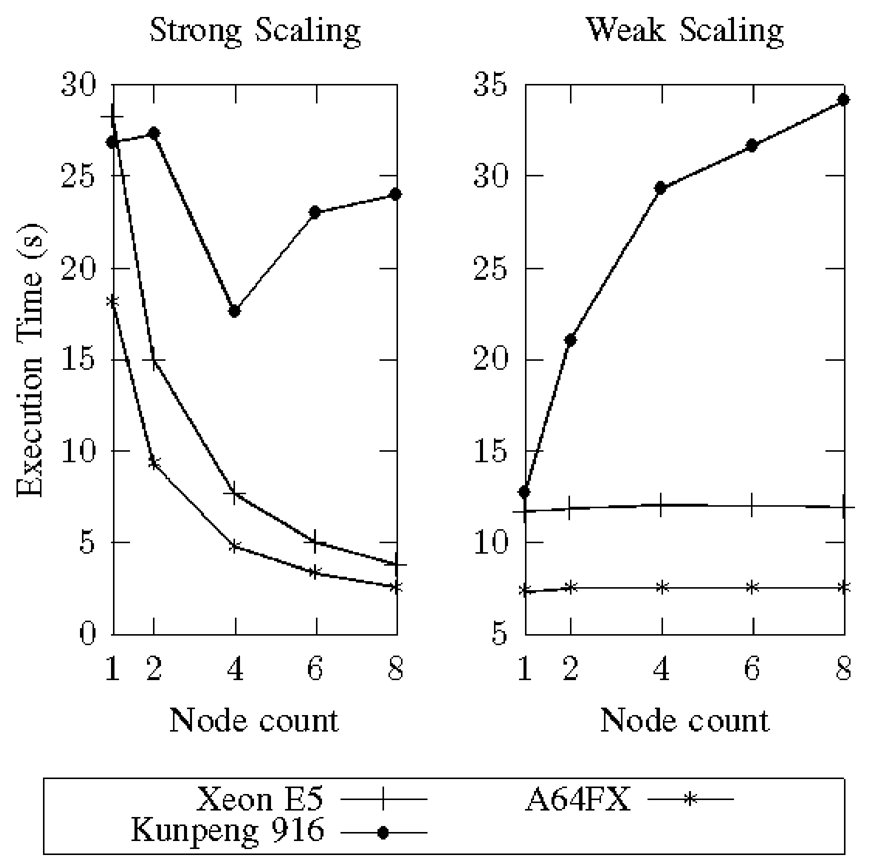


Fig. 3: 1D stencil: strong and weak scaling results. Strong scaling is done over 1.2 billion stencil points. Weak scaling is done by adding 480 million stencil points per node.

五、HPX在ARM机器上的测试

2D Stencil

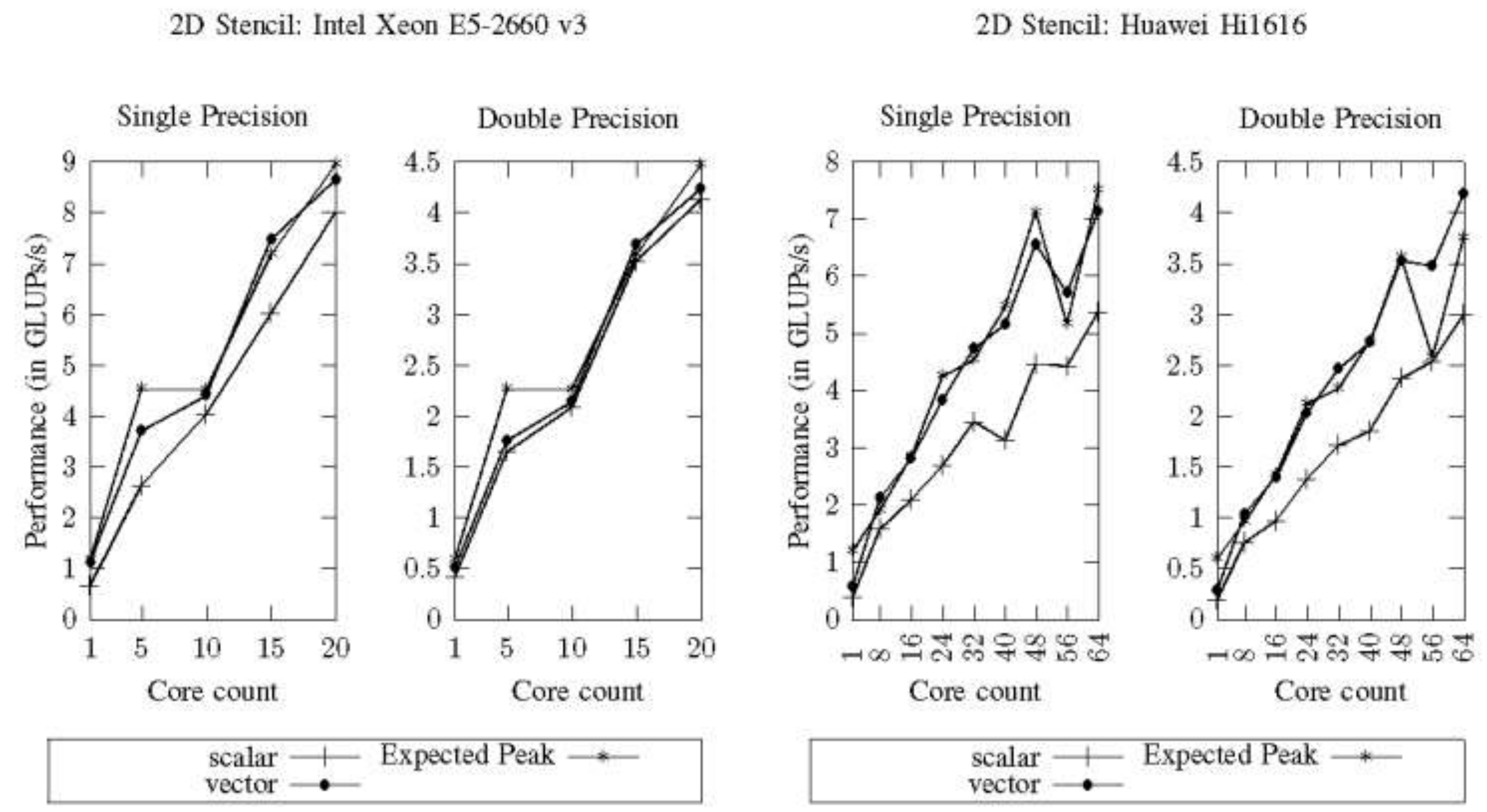


Fig. 4: 2D stencil: Results for Intel Xeon E5-2660 v3 with a grid size of 8192×131072 iterated over 100 time steps

Fig. 5: 2D stencil: Results for Huawei Kunpeng 916 with a grid size of 8192×131072 iterated over 100 time steps

五、HPX在ARM机器上的测试

2D Stencil

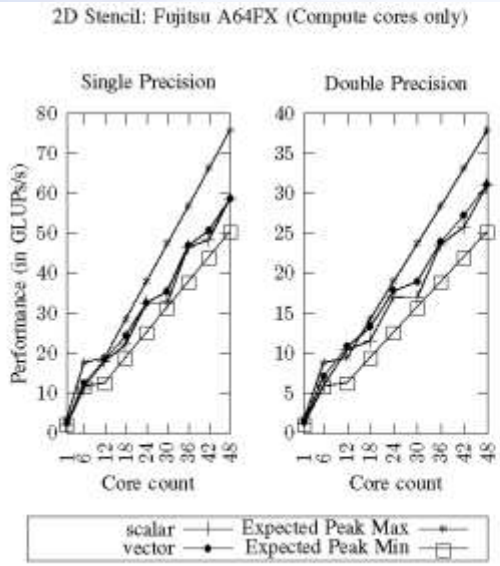


Fig. 6: 2D stencil: Results for Fujitsu A64FX with a grid size of 8192×131072 iterated over 100 time steps. Expected Peak Max assumes two memory transfers per iteration and Expected Peak Min assumes three memory transfers per iteration.

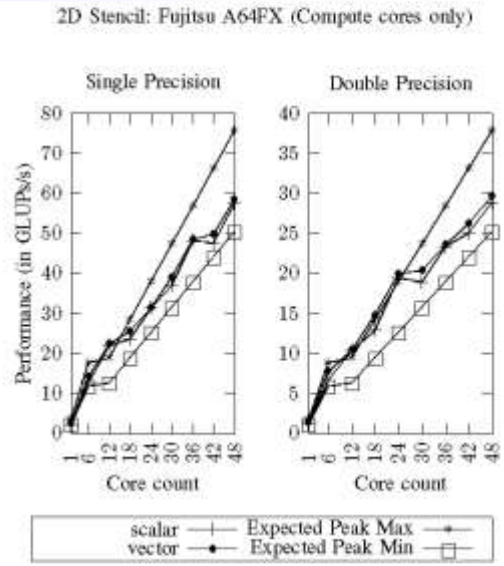


Fig. 7: 2D stencil: Results for Fujitsu A64FX with a grid size of 8192×196608 iterated over 100 time steps. Expected Peak Max assumes two memory transfers per iteration and Expected Peak Min assumes three memory transfers per iteration.

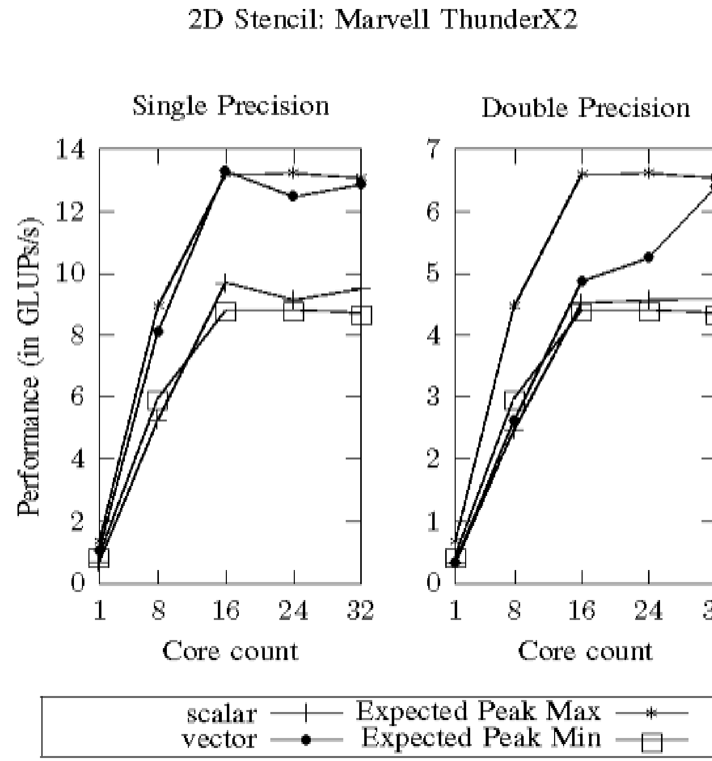


Fig. 8: 2D stencil: Results for Marvell ThunderX2 with a grid size of 8192×131072 iterated over 100 time steps. Expected Peak Max assumes two memory transfers per iteration and Expected Peak Min assumes three memory transfers per iteration.

参考文献: Performance Evaluation of ParalleX Execution model on Arm-based Platforms

HPX_async_mpi： MPI进程的异步计算与通信

https://gitee.com/lijian-cug/P2P_HPX_async_MPI int MPI_Isend(buf, count, datatype, rank, tag, comm, request);

代替

代替

hpx::future<int> f = hpx::async(executor, MPI_Isend, buf, count, datatype, rank, ta

声明

hpx::mpi::experimental::executor exec(MPI_COMM_WORLD);

形参替换顺序：

1 -> 3
2 -> 4
3 -> 5
4 -> 6
5 -> 7
6 -> executor
7 -> future

#include <hpx/modules/async_mpi.hpp>

针对非阻塞通信，使用executor和future参数代替

MPI_Comm comm和MPI_Request *request

```
// create an executor for MPI dispatch
hpx::mpi::experimental::executor exec(MPI_COMM_WORLD);

 // post an asynchronous receive using MPI_Irecv
hpx::future<int> f_recv = hpx::async(    exec, MPI_Irecv, &data, rank, MPI_INT, rank_from, i);

// attach a continuation to run when the recv completes,
f_recv.then([=, &tokens, &counter](auto&&){

     // call an application specific function
       msg_recv(rank, size, rank_to, rank_from, tokens[i], i);

    // send a new message
       hpx::future<int> f_send = hpx::async(exec, MPI_Isend, &tokens[i], 1, MPI_INT, rank_to, i);

    // when that send completes
       f_send.then([=, &tokens, &counter](auto&&)    {

    // call an application specific function
     msg_send(rank, size, rank_to, rank_from, tokens[i], i);
  });
}
```

int MPI_Isend(...);int MPI_Ibsend(...);int MPI_Issend(...);int MPI_Irsend(...);

int MPI_Irecv(...);int MPI_Imrecv(...);

int MPI_Ibarrier(...);

int MPI_Ibcast(...);

int MPI_Igather(...); int MPI_Igatherv(...);

int MPI_Iscatter(...); int MPI_Iscatterv(...);

int MPI_Iallgather(...); int MPI_Iallgatherv(...);

int MPI_Ialltoall(...); int MPI_Ialltoallv(...); int MPI_Ialltoallw(...);

int MPI_Ireduce(...); int MPI_Iallreduce(...);

int MPI_Ireduce_scatter(...);int MPI_Ireduce_scatter_block(...);

int MPI_Iscan(...);int MPI_Iexscan(...);

int MPI_Ineighbor_allgather(...);int MPI_Ineighbor_allgatherv(...);

int MPI_Ineighbor_alltoall(...);int MPI_Ineighbor_alltoallv(...);int MPI_Ineighbor_alltoallw(...);

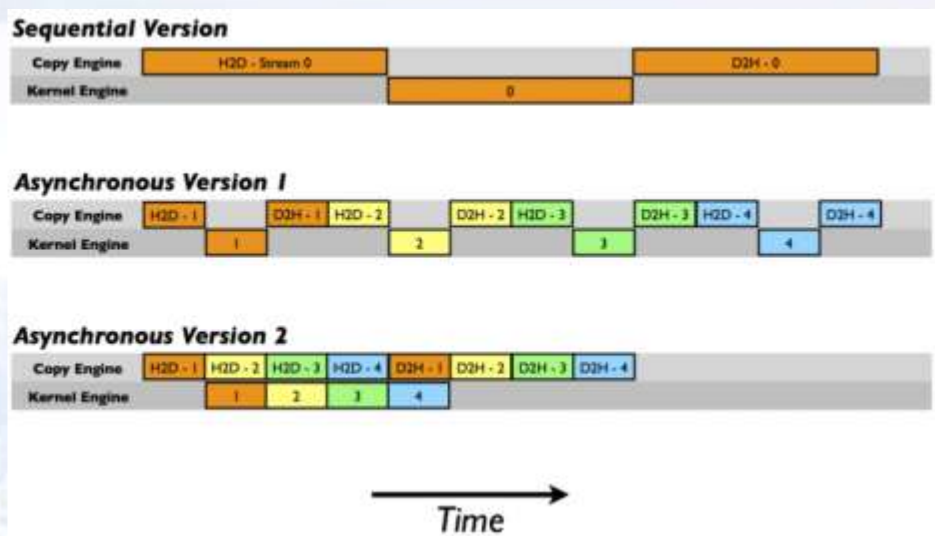
HPXCL：CPU-GPU之间的异步通信

```
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
            cudaMemcpyHostToDevice, stream[i]); kernel<<<streamSize/blockSize,
blockSize, 0, stream[i]>>>(d_a, offset);
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
cudaMemcpyDeviceToHost, stream[i]); }
```

串行模式

异步模式1

异步模式2



CUDA重叠核函数计算与数据传输（主机与设备间）

https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/

HPXCL介绍

Device

(1) 创建Buffer
(2) 创建Program

Buffer

(1) 显式地将数据拷贝到加速器
(2) 显式地拷贝数据to/from主机
(3) 可以向远程设备拷贝数据

Program

(1) 启动核函数
(2) 可以远程启动核函数

Patrick Diehl, Madhavan Seshadri, Thomas Heller, Hartmut Kaiser. 2018. Integration of CUDA Processing within the C++ library for parallelism and concurrency (HPX). arXiv:1810.11482v1

```
// 获取可使用的CUDA设备列表
std::vector<device> devices = get_all_devices(2, 0).get();

// 分配主机（CPU）上的数组
unsigned int* input;
unsigned int* n;
unsigned int* res;
cudaMallocHost((void**)&input, sizeof(unsigned int)* 1000);
cudaMallocHost((void**)&result,sizeof(unsigned int));
cudaMallocHost((void**)&n,sizeof(unsigned int));
memset (input,1,1000);
result[0] = 0;
n[0] = 1000;
```

```
// 创建设备，发现设备列表中的第1个设备
device cudaDevice = devices[0];

// 创建缓冲区，复制(CPU)数据进入设备(GPU)缓冲区
std::vector<hpx::lcos::future<void>> futures;

buffer outbuffer = cudaDevice.create_buffer(SIZE * sizeof(unsigned int)).get();
futures.push_back(outbuffer.enqueue_write(0, SIZE * sizeof(unsigned int), input));
buffer resbuffer = cudaDevice.create_buffer(sizeof(unsigned int)).get();
futures.push_back(resbuffer.enqueue_write(0,sizeof(unsigned int), result));
buffer lengthbuffer = cudaDevice.create_buffer(sizeof(unsigned int)).get();
futures.push_back(lengthbuffer.enqueue_write(0,sizeof(unsigned int), n));
```

首先使用cudaMalloc创建了3个缓冲区,然后使用cudaMemcpyAsync复制数据到缓冲区，该函数调用的future存储在一个future向量中，为后面的同步准备

// 使用NVCRT动态编译CUDA核函数

program prog = cudaDevice.create_program_with_file("kernel.cu").get(); // create_program_with_file or create_program_with_source

futures.push_back(prog.build("sum")); // the CUDA kernel is loaded from the file kernel.cu, the run time compilation of the kernel is started using NVRTC - CUDA Runtime Compilation, and the future is added to the vector of futures.

// 准备核函数的配置

hpx::cuda::server::program::Dim3 grid;

hpx::cuda::server::program::Dim3 block;

grid.x = grid.y = grid.z = 1;

block.x = 32;

block.y = block.z = 1; // the configuration of the kernel launch is defined.

```
// 设置核函数的形参:
std::vector<hpx::cuda::buffer>args;
args.push_back(outbuffer);
args.push_back(resbuffer);
args.push_back(lengthbuffer);

 // CUDA设备必须完成执行核函数，因此需要同步，执行hpx::wait_all，确保依赖都完成
hpx::wait_all(data_futures);
// 在默认的流上运行核函数
prog.run(args,"sum",grid,block).get();
// 拷贝计算结果到主机上，实际上使用的是cudaMemcpyAsync
unsigned int* res = resbuffer.enqueue_read_sync<unsigned int>(0,sizeof(unsigned int));
```

HPX源码获取、编译安装、使用CMake建立工程、配置和运行可以参考品HPX手册。

欢迎交流：

李健 email： jianli@cug.edu.cn