Applied Parallel Computing

parallel-computing.pro

OpenACC introduction

Aleksei Ivakhnenko

Contents

- Advantages of OpenACC
- Execution model
- OpenACC memory model
- Directive syntax in C and Fortran
  - Main directives
- Examples of **vector addition** and **reduction**
- OpenACC compilers:
  - PGI for NVIDIA and Radeon GPUs
  - GCC open-source compiler
  - Examples
- Hands-on on server
  - Connecting
  - Transferring
  - Reviewing examples

Links

- Specifications v.1.0 and 2.0 are available at:
  - http://www.openacc-standard.org
- Trial version of compiler
  - http://www.nvidia.com/object/openacc-toolkit.html#utm_source=shorturl&utm_medium=referrer&utm_campaign=openacctoolkit

OpenACC advantages

SAXPY example, C: OpenMP

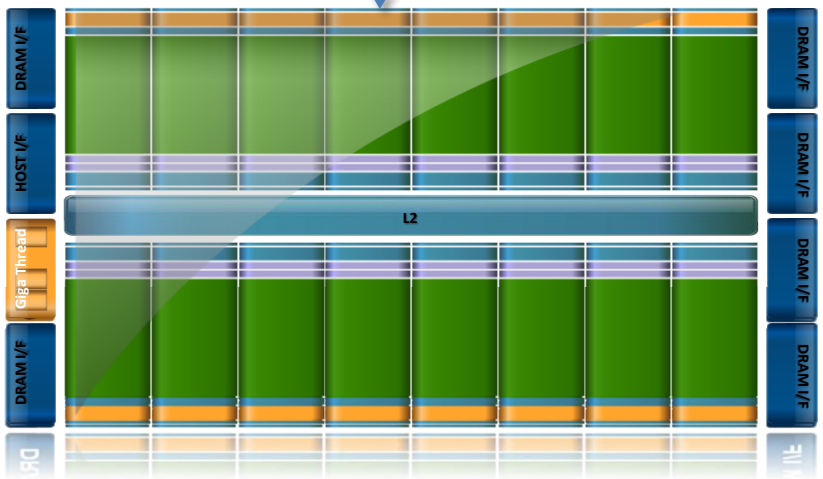
- Simple
- Open standard
- High performance

```
void saxpy(int n, float a, float *x,
           float *restrict y){
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY example, C: OpenACC

- Simple
- Open standard
- High performance




```
void saxpy(int n, float a, float *x,
           float *restrict y){
#pragma acc parallel
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

CUDA C/Fortran vs. OpenACC

- **CUDA C/Fortran:**
  - High performance of the manually optimized code
  - Porting code step by step
  - CUDA-capable GPU's only
  - One has to support 2 versions of code

- **OpenACC:**
  - Good performance is possible
  - Porting to different accelerators step by step
  - One can use not only CUDA-platforms
  - One version of code
  - Manual optimization is limited
  - Performance depends on the compiler "cleverness"

OpenACC API

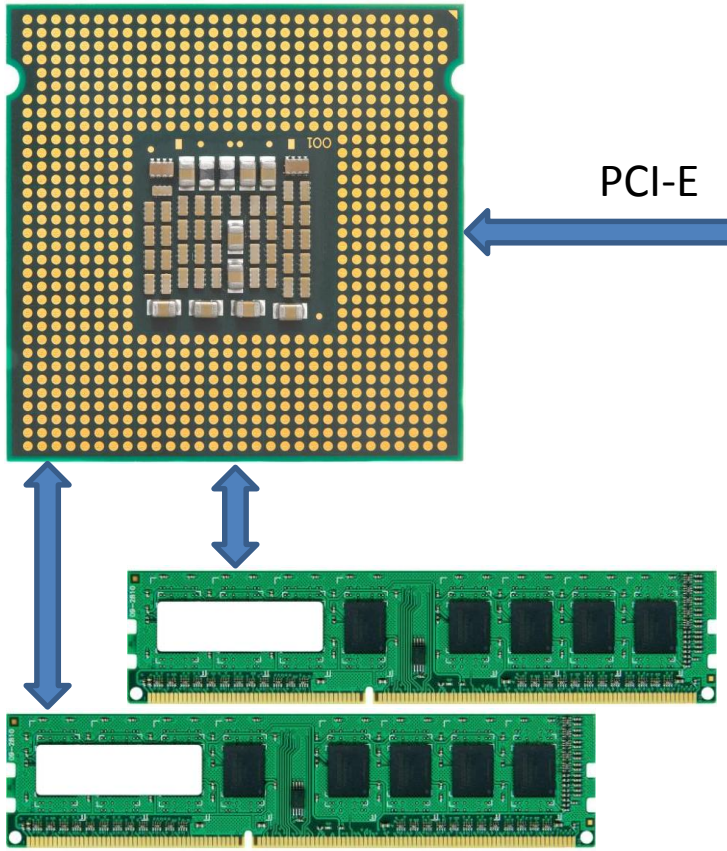
- **Directives point out parallel regions (C & Fortran)**
  - Offload parallel regions to GPU
  - Sources are cross-platform, cross-compiler and cross-accelerator
- **One can implement hybrid (CPU + ACC) high-level programs**
  - Without apparent accelerator initialization
  - Without apparent data management

OpenACC

- Programming model allows one to program easily, providing the compiler with hints:
  - Data management
  - Loops mapping
  - Other performance details
- Interoperation with other programming languages and libraries:
  - CUDA C / Fortran
  - GPU-accelerated libraries: CUFFT, CUBLAS, CUSPARSE, …

OpenACC Execution and Memory model

OpenACC Exectuion and Memory model



PCI-E

Host address space

Accelerator address space

OpenACC Execution model

- **CPU**
  - Executes the main part of the program
  - Allocates memory on the accelerator
  - Copies data from the host memory
  - Sends the code to the accelerator
  - Waits until the end of the kernel execution
  - Copies the results back to the host memory
  - Frees the accelerator memory
- **Accelerator**
  - Executes kernels
  - Can transfer data asynchronously with the execution

OpenACC Execution model

- 3 levels of parallelism: **gang, worker, vector**
- Mapped to an architecture as a set of processing elements (PEs)
- Each PE consists of workers, each worker is capable to execute vector instructions
- Mapping to an accelerator architecture is up to compiler (may be tracked via the output of compiler)

OpenACC API

Directive syntax

- Fortran

```
!$acc directive [clause [, clause] …]
        structured block
!$acc end directive
```

- C

```
#pragma acc directive [clause [, clause] …]
        structured block
```

- Compiling (with PGI compiler)

```
pgfortran -acc -Minfo=accel -ta=nvidia <filename>
pgcc –acc -Minfo=accel -ta=nvidia <filename>
```

Parallel construct

Fortran

```
!$acc parallel [clause [, clause]…]
        Structured block
!$acc end parallel
```

C

```
#pragma acc parallel [clause [, clause]…]
        Structured block
```

When the program encounters an accelerator parallel construct, one or more gangs are created to execute the accelerator parallel region

Parallel clauses

Main clauses

- if (condition)
- async [(exp)]
- num_gangs (exp)
- num_workers (exp)
- vector_length(exp)
- reduction(operator:list)

Data clauses

- copy*(list)
- create(list)
- present(list)
- present_or_copy*(list)
- present_or_create(list)
- deviceptr(list)
- private(list)
- firstprivate(list)

*<blank>|in|out

Restrictions

- Can’t have conditional entry- and leaving-points inside
- Should not depend on the clauses order
- Only one ‘if’ clause allowed
- Only the async, wait, num_gangs, num_workers, and vector_length clauses may follow a device_type clause

Kernels Construct

Fortran

```
!$acc kernels [clause [, clause]…]
        Structured block
!$acc end kernels
```

C

```
#pragma acc kernels [clause [, clause]…]
        Structured block
```

This construct defines a region of the program that is to be compiled into a sequence of kernels for execution on the accelerator device.

Kernels Construct

```
#pragma acc kernels
{
  for (int i= 0; i<n; i++)
  {
     for (int j = 0; j<n; j++)
     {
        a[i][j] = 0;
     }
  }
  for (int k = 0; k<n; k++)
  {
     b[k] = 1;
  }
}
```

Kernel 1

Kernel 2

The compiler will split the code in the kernels region into a sequence of accelerator kernels. Typically, each loop nest will be a distinct kernel.

Clauses

Main clauses

- if (condition)
- async [(exp)]

Data clauses

- copy*(list)
- create(list)
- present(list)
- present_or_copy*(list)
- present_or_create(list)
- deviceptr(list)
- private(list)
- firstprivate(list)

*<blank>|in|out

Loop Construct

Fortran

```
!$acc loop [clause [, clause]…]
        do loop
```

C

```
#pragma acc loop [clause [, clause]…]
        for loop
```

Clauses

- collapse(n)
- gang[(exp)]
- worker[(exp)]
- vector[(exp)]
- seq
- independent
- private(list)
- reduction(op:list)

Example: Vector addition

- Vector addition is “Hello world!” of parallel computing
- Takes 2 vectors (same size)
- Returns 1 resulting vector (same size)

$$C_0 = A_0 + B_0$$

$$C_1 = A_1 + B_1$$

$$\cdots\cdots\cdots\cdots\cdots\cdots$$

$$C_{n-1} = A_{n-1} + B_{n-1}$$

Example: Vector addition

```
int n=1000;
float a[n], b[n], c[n];
```

```
for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
    b[i]=(float)rand()/RAND_MAX;
}
```

Initialization

```
for (int i=0; i<n; i++)
{
    c[i]=a[i]+b[i];
}
```

Parallel region. Data cells are independent

$C_0 = A_0 + B_0$

$C_1 = A_1 + B_1$

. . . . . .

$C_{n-1} = A_{n-1} + B_{n-1}$

Example: Vector addition

```
int n=1000;
float a[n], b[n], c[n];
```

```
for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
    b[i]=(float)rand()/RAND_MAX;
}
```

Initialization

```
#pragma acc parallel loop independent
for (int i=0; i<n; i++)
{
    c[i]=a[i]+b[i];
}
```

Parallel region. Data cells are independent

$C_0 = A_0 + B_0$

$C_1 = A_1 + B_1$

. . . . . .

$C_{n-1} = A_{n-1} + B_{n-1}$

Example: Vector addition OpenMP

```
int n=1000;
float a[n], b[n], c[n];
```

```
for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
    b[i]=(float)rand()/RAND_MAX;
}
```

Initialization

```
#pragma omp target map (to:a,b), map (from:c)
#pragma omp parallel for
for (int i=0; i<n; i++)
{
    c[i]=a[i]+b[i];
}
```

Parallel region. Data cells are independent

$C_0 = A_0 + B_0$

$C_1 = A_1 + B_1$

. . . . . .

$C_{n-1} = A_{n-1} + B_{n-1}$

Example: Vector addition CUDA

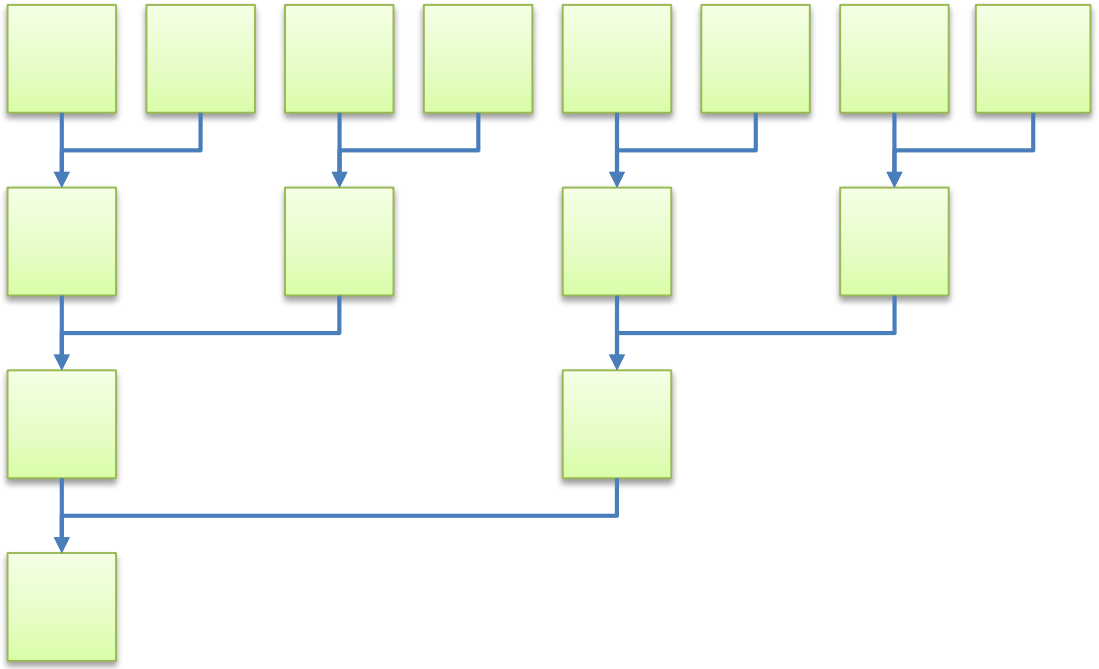
```
int n=1024;
float a[n], b[n], c[n];
float *da, *db, *dc;
cudaMalloc(da,sizeof(float)*n);
cudaMalloc(db,sizeof(float)*n);
cudaMalloc(dc,sizeof(float)*n);
for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
    b[i]=(float)rand()/RAND_MAX;
}
cudaMemcpy(da, a, sizeof(float)*n, cudaMemcpyHostToDevice);
cudaMemcpy(db, b, sizeof(float)*n, cudaMemcpyHostToDevice);
vecAdd<<<n/32,32>>>(da,db,dc,n);
cudaMemcpy(c, dc, sizeof(float)*n, cudaMemcpyDeviceToHost);
cudaFree(da);
cudaFree(db);
cudaFree(dc);
```

Example: Vector addition CUDA

```
__global__ void vecadd(float* a, float *b, float *c, int n)
{
    idx=threadIdx.x+blockIds.x*blockDim.x;
    if (idx<n)
    {
        c[idx]=a[idx]+b[idx];
    }
}
```

Example: Reduction

- Unlike vector addition, simple reduction takes only 1 vector and “operator”
- Returns 1 scalar
- Operator could be mathematical, logic operator, function (e.g. max) etc.
- Reduction is characterized by complex data dependency

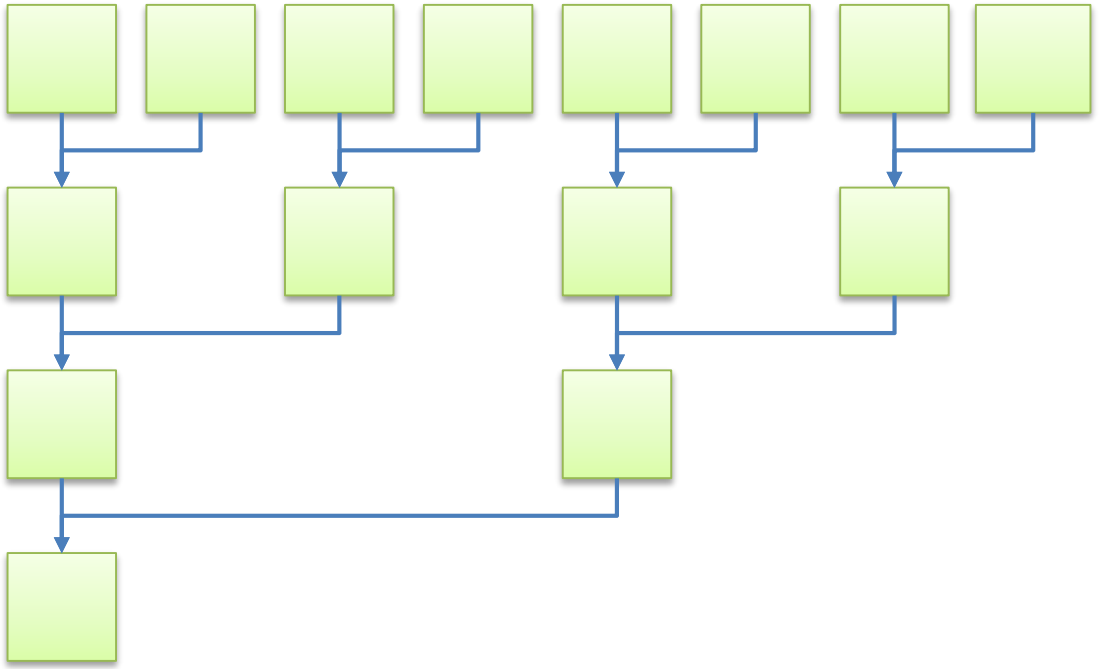


Example: Reduction

```
int n=1000;
float a[n];
float b=0;
```

```
for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
}
```

Initialization



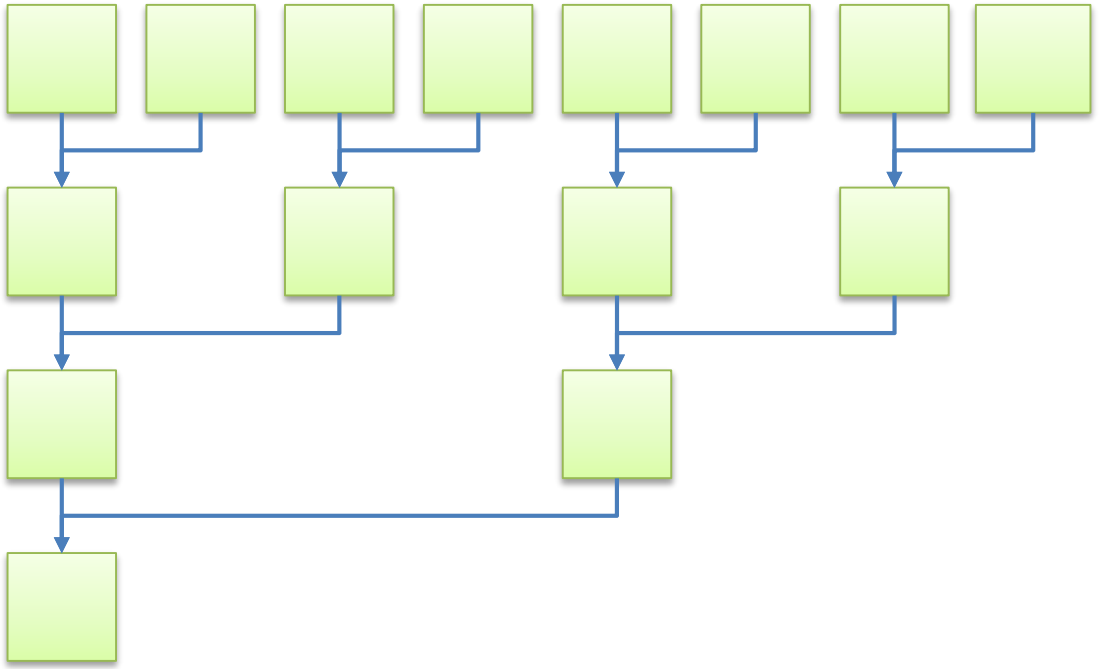
```
for (int i=0; i<n; i++)
{
    b+=a[i];
}
std::cout<<b[i]<<"\n";
```

Parallel region. Complex data dependency

Example: Reduction

```
int n=1000;
float a[n];
float b=0;
for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
}
```

Initialization



```
#pragma acc parallel reduction (+:b)
for (int i=0; i<n; i++)
{
    b+=a[i];
}
std::cout<<b[i]<<"\n";
```

Parallel region. Complex data dependency

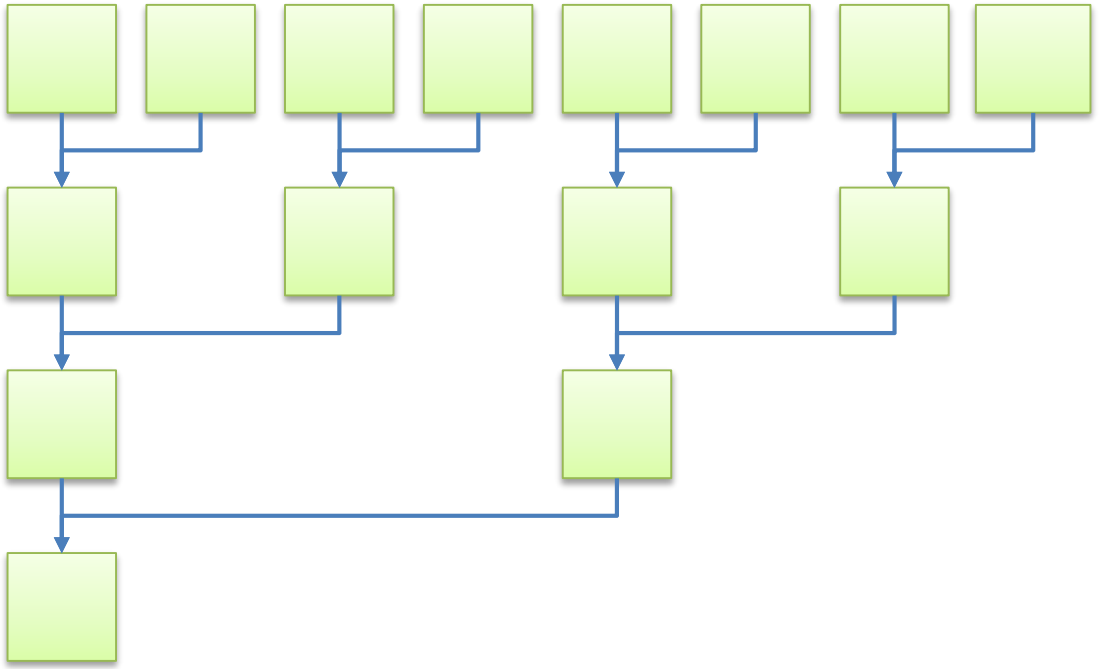
Example: Reduction OpenMP

```
int n=1000;
float a[n];
float b=0;
for (int i=0; i<n; i++)
{
    a[i]=(float)rand()/RAND_MAX;
}
```

Initialization

```
#pragma omp target map(to:a), map(from:b)
#pragma omp parallel for reduction(+:b)
for (int i=0; i<n; i++)
{
    b+=a[i];
}
std::cout<<b[i]<<"\n";
```

Parallel region. Complex data dependency



Example: Reduction CUDA

```
template <unsigned int blockSize>
__global__ void reduce(int *g_idata, int *g_odata, unsigned int n)
{
extern __shared__ int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + tid; unsigned int gridSize =
blockSize*2*gridDim.x; sdata[tid] = 0;
while (i < n)
{
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
if (blockSize >= 512)
{
    if (tid < 256)
    {
        sdata[tid] += sdata[tid + 256];
    }
    __syncthreads()
} if (blockSize >= 256)
{
    if (tid < 128)
    {
        sdata[tid] += sdata[tid + 128];
    } __syncthreads();
}
```

Example: Reduction CUDA

```
if (blockSize >= 128)
{
    if (tid <  64)
    {
        sdata[tid] += sdata[tid +  64];
    }
    __syncthreads();
}
if (tid < 32)
{
    if (blockSize >=  64)
        sdata[tid] += sdata[tid + 32];
    if (blockSize >=  32)
        sdata[tid] += sdata[tid + 16];
    if (blockSize >=  16)
        sdata[tid] += sdata[tid +  8];
    if (blockSize >=   8)
        sdata[tid] += sdata[tid +  4];
    if (blockSize >=   4)
        sdata[tid] += sdata[tid +  2];
    if (blockSize >=   2)
        sdata[tid] += sdata[tid +  1];
}
if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

OpenACC Compielrs

OpenACC complers

- There are different compilers for OpenACC code including PGI, PathScale, CRAY, CAPS.
- Despite OpenACC is an open standard, almost all compilers are non-free (even now after 4 years).
- GCC 5.0 introduced OpenACC support, but it's very limited yet.
- PGI (owned by NVIDIA) is one of the most advanced compilers. Not so long ago it received free academic license and 90 day trial license.

Parallel vs. Kernels

```
int a [1000];
int b [1000];
#pragma acc parallel
{
  for (int i=0; i<1000; i++)
    a[i] = i – 100 + 23;

  for (int j=0; j<1000; j++)
    b[j] = j – 10 + 213;
}
```

Parallel vs. Kernels

```
pgcc parallel.c -acc -Minfo=accel -ta=nvidia,time -o parallel
parallel.c:
main:
     10, Accelerator kernel generated
         Generating Tesla code
         12, #pragma acc loop vector(128) /* threadIdx.x */
         16, #pragma acc loop vector(128) /* threadIdx.x */
     10, Generating copyout(a[:],b[:])
     12, Loop is parallelizable
     16, Loop is parallelizable
```

Parallel vs. Kernels

```
int a [1000];
int b [1000];
#pragma acc kernels
{
  for (int i=0; i<1000; i++)
    a[i] = i – 100 + 23;

  for (int j=0; j<1000; j++)
    b[j] = j – 10 + 213;
}
```

Parallel vs. Kernels

```
pgcc kernels.c -acc -Minfo=accel -ta=nvidia,time -o kernels
kernels.c:
main:
     10, Generating copyout(a[:],b[:])
     12, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         12, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     16, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         16, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

GCC limitations

- GCC 5 includes a preliminary implementation of the OpenACC2.0a specification.
- The execution model currently only allows for one gang, one worker, and a number of vectors. These vectors will all execute in "vector-redundant" mode. This means that inside a parallel construct, offloaded code outside of any loop construct will be executed by all vectors, not just a single vector. The reduction clause is not yet supported with the parallel construct.
- The kernels construct so far is supported only in a simplistic way: the code will be offloaded, but execute with just one gang, one worker, one vector. No directives are currently supported inside kernels constructs. Reductions are not yet supported inside kernels constructs.
- The atomic, cache, declare, host_data, and routine directives are not yet supported.
- The default(none), device_type, firstprivate, and private clauses are not yet supported. A parallel construct's implicit data attributes for scalar data types will be treated as present_or_copyinstead of firstprivate. Only the collapse clause is currently supported for loop constructs, and there is incomplete support for the reduction clause.
- Combined directives (kernels loop, parallel loop) are not yet supported; use kernels alone, or parallel followed by loop, instead.
- Nested parallelism (cf. CUDA dynamic parallelism) is not yet supported.
- Usage of OpenACC constructs inside multithreaded contexts (such as created by OpenMP, or pthread programming) is not yet supported.
- https://gcc.gnu.org/wiki/OpenACC

Parallel (gcc)

```
int a [1000];
int b [1000];
#pragma acc parallel
{
  for (int i=0; i<1000; i++)
    a[i] = i – 100 + 23;

  for (int j=0; j<1000; j++)
    b[j] = j – 10 + 213;
}
```

Parallel vs. Kernels

```
/opt/gcc-5.2.0_offload/usr/local/bin/gcc parallel.c -fopenacc -o parallel
aivahnenko@tesla-cmc:/scratch/aivahnenko/openacc/gcc/examples/parallel$ ./parallel
libgomp: num_gangs (4) different from one is not yet supported
```

- GCC now only starts OpenACC support so it can be used for studying OpenACC, but doesn't give any good performance.
- GCC has a lot of restrictions and can't be used for big projects yet.

Independent vs. Seq

```
int a [10000];

#pragma acc kernels
{
    #pragma acc loop independent
    for (int i=0; i<100; i++)
    {
      #pragma acc loop independent
      for (int j=0; j<100; j++)
          a[i*100 + j] = i - 100 + 23 + j;
    }
}
```

Independent vs. Seq

```
pgcc independent.c -acc -Minfo=accel -ta=nvidia,time -o independent
independent.c:
main:
      9, Generating copy(a[:])
     12, Loop is parallelizable
     15, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         12, #pragma acc loop gang /* blockIdx.y */
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Independent vs. Seq

```
int a [10000];

#pragma acc kernels
{
    #pragma acc loop independent
        for (int i=0; i<100; i++)
        {
            #pragma acc loop seq
            for (int j=0; j<100; j++)
                a[i*100 + j] = i - 100 + 23 + j;
        }
}
```

Independent vs. Seq

```
pgcc seq.c -acc -Minfo=accel -ta=nvidia,time -o seq
seq.c:
main:
      9, Generating copy(a[:])
     12, Loop is parallelizable
     15, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         12, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Example

Example

```
//Jacobi solver
while (eps> tolerance)
{
    change = 0.0f;
    iter++;
//Parallel region to be executed on GPU
        for (int j = 1; j<n-1; j++)
        {
            for (int i = 1, i<n-1; i++)
            {
                newa[i][j]= w0 * a[i][j] +
                    w1 * (a[i-1][j] +a[i][j-1]+a[i+1][j]+a[i][j+1]) +
                    w2 * (a[i-1][j-1]+a[i-1][j+1]+a[i+1][j-1]+a[i+1][j+1]);
               eps = max(eps, abs(newa[i][j] – a[i][j]);
            }
        }
        swap(a,newa);
//end of parallel region
}
```

Example

```
//Jacobi solver
while (eps> tolerance)
{
    change = 0.0f;
    iter++;
#pragma acc parallel
{
        for (int j = 1; j<n-1; j++)
        {
            for (int i = 1, i<n-1; i++)
            {
                newa[i][j]= w0 * a[i][j] +
                    w1 * (a[i-1][j] +a[i][j-1]+a[i+1][j]+a[i][j+1]) +
                    w2 * (a[i-1][j-1]+a[i-1][j+1]+a[i+1][j-1]+a[i+1][j+1]);
               eps = max(eps, abs(newa[i][j] – a[i][j]);
            }
        }
        swap(a,newa);
}
}
```

Example

```
#pragma acc data copy(a, newa)
{
while (eps> tolerance)
{
    change = 0.0f;
    iter++;
#pragma acc parallel reduction (max:eps)
{
        for (int j = 1; j<n-1; j++)
        {
            for (int i = 1, i<n-1; i++)
            {
                newa[i][j]= w0 * a[i][j] +
                    w1 * (a[i-1][j] +a[i][j-1]+a[i+1][j]+a[i][j+1]) +
                    w2 * (a[i-1][j-1]+a[i-1][j+1]+a[i+1][j-1]+a[i+1][j+1]);
               eps = max(eps, abs(newa[i][j] – a[i][j]);
            }
        }
        swap(a,newa);
}
}
}
```

Example

```
#pragma acc data copyin(a) create(newa)
{
while (eps> tolerance)
{
    change = 0.0f;
    iter++;
#pragma acc parallel reduction (max:eps)
{
        for (int j = 1; j<n-1; j++)
        {
            for (int i = 1, i<n-1; i++)
            {
                newa[i][j]= w0 * a[i][j] +
                    w1 * (a[i-1][j] +a[i][j-1]+a[i+1][j]+a[i][j+1]) +
                    w2 * (a[i-1][j-1]+a[i-1][j+1]+a[i+1][j-1]+a[i+1][j+1]);
               eps = max(eps, abs(newa[i][j] – a[i][j]);
            }
        }
        swap(a,newa);
}
}
}
```

Example

```
#pragma acc data copyin(a) create(newa)
{
while (eps> tolerance)
{
    change = 0.0f;
    iter++;
#pragma acc parallel reduction (max:eps), vector_length (256)
{
        for (int j = 1; j<n-1; j++)
        {
            for (int i = 1, i<n-1; i++)
            {
                newa[i][j]= w0 * a[i][j] +
                    w1 * (a[i-1][j] +a[i][j-1]+a[i+1][j]+a[i][j+1]) +
                    w2 * (a[i-1][j-1]+a[i-1][j+1]+a[i+1][j-1]+a[i+1][j+1]);
               eps = max(eps, abs(newa[i][j] – a[i][j]);
            }
        }
        swap(a,newa);
}
}
}
```

Example

Tesla T10 Processor

$ ./jacobi.acc 1024

reached delta= 0.09998 in 3430 iterations for 1024 x 1024 array

time = 25.8760 seconds

Intel(R) Xeon(R) CPU E5620 @2.40GHz

$ ./jacobi 1024

reached delta= 0.09998 in 3430 iterations for 1024 x 1024 array

time(host) = 140.386185 seconds

Example

	N=400	N=512	N=1024
CPU One-thread	6.7759	16.0250	140.3861
OpenMP	1.8580	3.7771	29.6452
PGI OpenACC	6.8860	8.9890	9.2995
CUDA C (manually optimized)	3.8095	4.1140	6.5899

Questions?

Aleksei Ivakhnenko
ivakhnenko@parallel-computing.pro

Useful links:

http://parallel-compute.com
http://www.openacc-standard.org
http://www.pgroup.com/resources/accel.htm
http://developer.nvidia.com/category/zone/cuda-zone