L8179 - ZERO TO GPU HERO WITH OPENACC

Jeff Larkin, GTC 2019, March 2019



OUTLINE Topics to be covered

- What is OpenACC
- Profile-driven Development
- OpenACC with CUDA Unified Memory
- OpenACC Data Directives
- OpenACC Loop Optimizations
- Where to Get Help



ABOUT THIS SESSION

- The objective of this session is to give you a brief introduction of OpenACC programming for NVIDIA GPUs
- This is an instructor-led session, there will be no hands on portion
- For hands on experience, please consider attending DLIT903 OpenACC 2X in 4 Steps or L9112 - Programming GPU-Accelerated POWER Systems with OpenACC if your badge allows
- Feel free to interrupt with questions



INTRODUCTION TO OPENACC



OpenACC is a directivesbased programming approach to parallel computing designed for performance and portability on CPUs and GPUs for HPC.





3 WAYS TO ACCELERATE APPLICATIONS



OPENACC PORTABILITY

Describing a generic parallel machine

- OpenACC is designed to be portable to many existing and future parallel platforms
- The programmer need not think about specific hardware details, but rather express the parallelism in generic terms
- An OpenACC program runs on a host (typically a CPU) that manages one or more parallel devices (GPUs, etc.). The host and device(s) are logically thought of as having separate memories.





OPENACC Three major strengths

Incremental	Single Source	e L	ow Learning Curve



Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

```
Begin with a working
                                         sequential code.
    Enhance Sequential Code
#pragma acc parallel loop
for(i = 0; i < N; i++)
{
    < loop code >
                                   Parallelize it with OpenACC.
}
#pragma acc parallel loop
for( i = 0; i < N; i++ )</pre>
{
                                     Rerun the code to verify
    < loop code >
                                         correct behavior,
}
                                     remove/alter OpenACC
```

code as needed.





Single Source	Low Learning Curve



Supported Platforms POWER Sunway x86 CPU x86 Xeon Phi NVIDIA GPU PEZY-SC

Single Source Rebuild the same code on multiple architectures Compiler determines how to parallelize for the desired machine Sequential code is maintained

The compiler can **ignore** your OpenACC code additions, so the same code can be used for **parallel** or **sequential** execution.

```
int main(){
    ...
    #pragma acc parallel loop
    for(int i = 0; i < N; i++)
        < loop code >
}
```



Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

Single Source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

Low Learning Curve





The programmer will give hints to the compiler about which parts of the code to parallelize.

The compiler will then generate parallelism for the target parallel hardware.

Low Learning Curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.



Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

Single Source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

Low Learning Curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.



OPENACC SUCCESSES



OPENACC SYNTAX



OPENACC SYNTAX

Syntax for using OpenACC directives in code

C/C++			
<pre>#pragma <code></code></pre>	acc	directive	clauses

Fortran			
		rtron	
	ΓU	Illall	

!\$acc directive clauses
<code>

- A pragma in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.
- A directive in Fortran is a specially formatted comment that likewise instructions the compiler in it compilation of the code and can be freely ignored.
- "*acc*" informs the compiler that what will come is an OpenACC directive
- Directives are commands in OpenACC for altering our code.
- Clauses are specifiers or additions to directives.



EXAMPLE CODE



LAPLACE HEAT TRANSFER

Introduction to lab code - visual

We will observe a simple simulation of heat distributing across a metal plate.

We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.





EXAMPLE: JACOBI ITERATION

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
- Common, useful algorithm





JACOBI ITERATION: C CODE

```
while ( err > tol && iter < iter_max ) {
    err=0.0;</pre>
```

```
for( int j = 1; j < n-1; j++) {
   for(int i = 1; i < m-1; i++) {</pre>
```

```
Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j-1][i] + A[j-1][i] + A[j+1][i]);
```

```
err = max(err, abs(Anew[j][i] - A[j][i]));
}
```

```
for( int j = 1; j < n-1; j++) {
  for( int i = 1; i < m-1; i++ ) {
     A[j][i] = Anew[j][i];
  }
}</pre>
```



OpenACC

}



Iterate until converged

Compute max error for convergence

Swap input/output arrays

PROFILE-DRIVEN DEVELOPMENT



OPENACC DEVELOPMENT CYCLE

- Analyze your code to determine most likely places needing parallelization or optimization.
- Parallelize your code by starting with the most time consuming parts and check for correctness.
- Optimize your code to improve observed speed-up from parallelization.





PROFILING SEQUENTIAL CODE

Profile Your Code

Obtain detailed information about how the code ran.

This can include information such as:

Total runtime

OpenACC

- Runtime of individual routines
- Hardware counters

Identify the portions of code that took the longest to run. We want to focus on these "hotspots" when parallelizing.



PROFILING SEQUENTIAL CODE

First sight when using PGPROF

- Profiling a simple, sequential code
- Our sequential program will on run on the CPU
- To view information about how our code ran, we should select the "CPU Details" tab





PROFILING SEQUENTIAL CODE CPU Details

- Within the "CPU Details" tab, we can see the various parts of our code, and how long they took to run
- We can reorganize this info using the three options in the top-right portion of the tab
- We will expand this information, and see more details about our code

File View Window	v <u>R</u> un Help						
😁 🗟 🖳 📑 🖷	9 ° ₉ • I 🕂 🗨 🤤	R F K		\$ ₹			
💺 laplace.nvvp 🛿							- 0
	0 s	0.1 s	0.2 s	0.3 s	0.4 s	0.5 s	0.6 s
🗔 Analysis 🔳 GPU	Details 🛅 CPU Deta	ils ដ 🗉 Co	onsole 🗔 Setti	ngs 🗈 🖬		Properties 🖾	- 0
-							
TOTAL	Use the buttons or	h the top-rig	ht of this view	to select how t	o display prc	Select or highlight	a single
Event	%	Time	2			interval to see pro	perties
/home/ewright/e	edited_laplac 53.3259	6 21.2	5 s				
/opt/pgi/linux86	-64/17.4/lib/l 46.55%	18.5	5 s				
/lib/x86_64-linux	-gnu/libc-2.2. 0.125%	0.05	s				
					2.0		



PROFILING SEQUENTIAL CODE CPU Details

- We can see that there are two places that our code is spending most of its time
- 21.49 seconds in the "calcNext" function
- 19.04 seconds in a memcpy function
- The c_mcopy8 that we see is actually a compiler optimization that is being applied to our "swap" function

+ opiecentrip		50000000 c	100000000 c	150000000 c	200000000 c	350000000
🖩 Analysis 🔚 GPU Details 🎛	CPU Details ¤	🗳 Console 🗔 Settings	‡h:	\$↓ 0	□ Properties ⊠	
TOTAL Use the b	outtons on the t	op-right of this view to se	elect how to display pr	rofile data More	Select or highlight a single in properties	nterval to see
/home/ewright/edited_lapla	c 21 519%	21.51 s				
✓ /laplace2d c	21.519%	21.51 5				
	21.519%	21.515				
calcNext:37	21.499%	21.49 s				
calcNext:35	0.02%	0.02 s				
/opt/pgi/linux86-64/17.4/lib,	/l 19.048%	19.04 s				
 Unknown Filename 	19.048%	19.04 s				
▼ c mcopy8	19.048%	19.04 s				
c_mcopy8:-1	19.048%	19.04 s				
/lib/x86_64-linux-gnu/libc-2.3	2. 0.06%	0.06 s				



PROFILING SEQUENTIAL CODE

- We are also able to select the different elements in the CPU Details by double-clicking to open the associated source code
- Here we have selected the "calcNext:37" element, which opened up our code to show the exact line (line 37) that is associated with that element







Expressing parallelism

```
#pragma acc parallel
{
```

When encountering the *parallel* directive, the compiler will generate *1 or more parallel gangs*, which execute redundantly.









Parallelizing a single loop

C/C++

```
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; j < N; i++)
        a[i] = 0;</pre>
```

Fortran

OpenACC

```
!$acc parallel
 !$acc loop
 do i = 1, N
     a(i) = 0
 end do
!$acc end parallel
```



- This parallel region is marked by curly braces in C/C++ or a start and end directive in Fortran
- The loop directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs

Parallelizing a single loop

C/C++

#pragma	acc	parallel loop
<pre>for(int</pre>	i =	<pre>0; j < N; i++)</pre>
a[i] =	= <mark>0</mark> ;	

Fortran

!\$acc parallel loop
do i = 1, N
 a(i) = 0
end do

- This pattern is so common that you can do all of this in a single line of code
- In this example, the parallel loop directive applies to the next loop
- This directive both marks the region for parallel execution and distributes the iterations of the loop.
- When applied to a loop with a data dependency, parallel loop may produce incorrect results







Parallelizing many loops

```
#pragma acc parallel loop
for(int i = 0; i < N; i++)
a[i] = 0;
#pragma acc parallel loop
for(int j = 0; j < M; j++)
b[j] = 0;</pre>
```

- To parallelize multiple loops, each loop should be accompanied by a parallel directive
- Each parallel loop can have different loop boundaries and loop optimizations
- Each parallel loop can be parallelized in a different way
- This is the recommended way to parallelize multiple loops. Attempting to parallelize multiple loops within the same parallel region may give performance issues or unexpected results


PARALLELIZE WITH OPENACC PARALLEL LOOP

```
while ( err > tol && iter < iter_max ) {
    err=0.0;</pre>
```

Parallelize first loop nest, max *reduction* required.



#pragma acc parallel loop
 for(int j = 1; j < n-1; j++) {
 for(int i = 1; i < m-1; i++) {
 A[j][i] = Anew[j][i];
 }
 }
 iter++;
 }
OpenACC @</pre>

We didn't detail *how* to parallelize the loops, just *which* loops to parallelize.

BUILDING THE CODE (GPU)

\$ pgcc -fast -ta=tesla:managed -Minfo=accel laplace2d_uvm.c main:

- 63, Accelerator kernel generated Generating Tesla code 64, #pragma acc loop gang /* blockIdx.x */ Generating reduction(max:error) 66, #pragma acc loop vector(128) /* threadIdx.x */ 63, Generating implicit copyin(A[:]) Generating implicit copyout(Anew[:]) Generating implicit copy(error) 66, Loop is parallelizable 74, Accelerator kernel generated Generating Tesla code 75, #pragma acc loop gang /* blockIdx.x */ 77, #pragma acc loop vector(128) /* threadIdx.x */ 74, Generating implicit copyin(Anew[:]) Generating implicit copyin(Anew[:])
 - 77, Loop is parallelizable



BUILDING THE CODE (MULTICORE)

\$ pgcc -fast -ta=multicore -Minfo=accel laplace2d_uvm.c main:

- 63, Generating Multicore code
 - 64, #pragma acc loop gang
- 64, Accelerator restriction: size of the GPU copy of Anew, A is unknown Generating reduction(max:error)
- 66, Loop is parallelizable
- 74, Generating Multicore code
 - 75, #pragma acc loop gang
- 75, Accelerator restriction: size of the GPU copy of Anew, A is unknown
- 77, Loop is parallelizable



OPENACC SPEED-UP



BUILDING THE CODE (GPU)

\$ pgcc -fast -ta=tesla -Minfo=accel laplace2d_uvm.c PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index for symbol (laplace2d_uvm.c: 63) PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index for symbol (laplace2d_uvm.c: 74) main:

63, Accelerator kernel generated

Generating Tesla code

- 63, Generating reduction(max:error)
- 64, #pragma acc loop gang /* blockIdx.x */
- 66, #pragma acc loop vector(128) /* threadIdx.x */
- 64, Accelerator restriction: size of the GPU copy of Anew, A is unknown
- 66, Loop is parallelizable
- 74, Accelerator kernel generated Generating Tesla code
 - 75, #pragma acc loop gang /* blockIdx.x */
 - 77, #pragma acc loop vector(128) /* threadIdx.x */
- 75, Accelerator restriction: size of the GPU copy of Anew, A is unknown
- 77, Loop is parallelizable



OPTIMIZE DATA MOVEMENT



EXPLICIT MEMORY MANAGEMENT

- Many parallel accelerators (such as devices) have a separate memory pool from the host
- These separate memories can become out-of-sync and contain completely different data
- Transferring between these two memories can be a very time consuming process





OPENACC DATA DIRECTIVE Definition

- The data directive defines a lifetime for data on the device
- During the region data should be thought of as residing on the accelerator
- Data clauses allow the programmer to control the allocation and movement of data

#pragma acc data <i>clauses</i> {	
< Sequential and/or Parallel code	>
}	

!\$acc data clauses

< Sequential and/or Parallel code >

!\$acc end data



DATA CLAUSES

copy(*list*) Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

Principal use: For many important data structures in your code, this is a logical default to input, modify and return the data.

copyin(*list*) Allocates memory on GPU and copies data from host to GPU when entering region.

Principal use: Think of this like an array that you would use as just an input to a subroutine.

copyout(*list*) Allocates memory on GPU and copies data to the host when exiting region.

Principal use: A result that isn't overwriting the input data structure.

create(*list* **)** Allocates memory on GPU but does not copy.

OpenACC

Principal use: Temporary arrays.

ARRAY SHAPING

- Sometimes the compiler needs help understanding the shape of an array
- The first number is the start index of the array
- In C/C++, the second number is how much data is to be transferred
- In Fortran, the second number is the ending index





ARRAY SHAPING (CONT.)

Multi-dimensional Array shaping

copy(array[0:N][0:M])

Both of these examples copy a 2D array to the device

copy(array(1:N, 1:M))

Fortran

C/C++



ARRAY SHAPING (CONT.)

Partial Arrays

copy(array[i*N/4:N/4])

C/C++

Both of these examples copy only 1/4 of the full array

copy(array(i*N/4:i*N/4+N/4))

Fortran



STRUCTURED DATA DIRECTIVE

Example

OpenACC





OPTIMIZED DATA MOVEMENT

```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
    err=0.0;</pre>
```



Copy A to/from the accelerator only when needed.

Copy initial condition of Anew, but not final value

}

}

REBUILD THE CODE

pgcc -fast -ta=tesla -Minfo=accel laplace2d_uvm.c
main:

- 60, Generating copy(A[:m*n])
 Generating copyin(Anew[:m*n])
- 64, Accelerator kernel generated Generating Tesla code
 - 64, Generating reduction(max:error)
 - 65, #pragma acc loop gang /* blockIdx.x */
 - 67, #pragma acc loop vector(128) /* threadIdx.x */
- 67, Loop is parallelizable
- 75, Accelerator kernel generated

Generating Tesla code

- 76, #pragma acc loop gang /* blockIdx.x */
- 78, #pragma acc loop vector(128) /* threadIdx.x */
- 78, Loop is parallelizable



Now data movement only happens at our data region.



OPENACC SPEED-UP



DATA SYNCHRONIZATION



OPENACC UPDATE DIRECTIVE

update: Explicitly transfers data between the host and the device

Useful when you want to synchronize data in the middle of a data region Clauses:

self: makes host data agree with device data

device: makes device data agree with host data

#pragma acc update self(x[0:count])
#pragma acc update device(x[0:count])

C/C++

!\$acc update self(x(1:end_index))
!\$acc update device(x(1:end_index))
Fortran



OPENACC UPDATE DIRECTIVE





#pragma acc update self(A[0:N])

SYNCHRONIZE DATA WITH UPDATE

```
int* allocate_array(int N){
    int* A=(int*) malloc(N*sizeof(int));
    #pragma acc enter data create(A[0:N])
    return A;
}
```

```
void deallocate_array(int* A){
    #pragma acc exit data delete(A)
    free(A);
}
```

```
void initialize_array(int* A, int N){
    for(int i = 0; i < N; i++){
        A[i] = i;
    }
    #pragma acc update device(A[0:N])</pre>
```

- Inside the initialize function we alter the host copy of 'A'
- This means that after calling initialize the host and device copy of 'A' are out-of-sync
- We use the update directive with the device clause to update the device copy of 'A'
- Without the update directive later compute regions will use incorrect data.



FURTHER OPTIMIZATIONS



PROFILING GPU CODE (PGPROF)

Using PGPROF to profile GPU code

- PGPROF presents far more information when running on a GPU
- We can view CPU Details, GPU Details, a Timeline, and even do Analysis of the performance





PROFILING GPU CODE (PGPROF)

Using PGPROF to profile GPU code

- MemCpy(HtoD): This includes data transfers from the Host to the Device ' (CPU to GPU)
- MemCpy(DtoH): These are data transfers from the Device to the Host (GPU to CPU)
- Compute: These are our computational functions. We can see our calcNext and swap function





LOOP OPTIMIZATIONS



COLLAPSE CLAUSE

- collapse(N)
- Combine the next N tightly nested loops
- Can turn a multidimensional loop nest into a single-dimension loop
- This can be extremely useful for increasing memory locality, as well as creating larger loops to expose more parallelism

<pre>#pragma acc parallel loop collapse(2)</pre>
for(i = 0; i < size; i++)
for(j = 0; j < size; j++)
<pre>double tmp = 0.0f;</pre>
<pre>#pragma acc loop reduction(+:tmp)</pre>
<pre>for(k = 0; k < size; k++)</pre>
tmp += a[i][k] * b[k][j];
c[i][j] = tmp;



COLLAPSE CLAUSE

collapse(2)

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

#pragma acc parallel loop collapse(2)
for(i = 0; i < 4; i++)
for(j = 0; j < 4; j++)
array[i][j] = 0.0f;</pre>



TILE CLAUSE

- tile (x , y , z, ...)
- Breaks multidimensional loops into "tiles" or "blocks"
- Can increase data locality in some codes
- Will be able to execute multiple "tiles" simultaneously

<pre>#pragma acc kernels loop tile(32, 32)</pre>			
<pre>for(i = 0; i < size; i++)</pre>			
for(j = 0; j < size; j++)			
<pre>for(k = 0; k < size; k++)</pre>			
c[i][j] += a[i][k] * b[k][j];			



TILE CLAUSE

```
#pragma acc kernels loop tile(2,2)
for(int x = 0; x < 4; x++){
   for(int y = 0; y < 4; y++){
      array[x][y]++;
   }
}</pre>
```

tile (2 , 2)





GANG WORKER VECTOR

- Gang / Worker / Vector defines the various levels of parallelism we can achieve with OpenACC
- This parallelism is most useful when parallelizing multi-dimensional loop nests
- OpenACC allows us to define a generic Gang / Worker / Vector model that will be applicable to a variety of hardware, but we fill focus a little bit on a GPU specific implementation





OPTIMIZED LOOP

```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
    err=0.0;</pre>
```

```
#pragma acc parallel loop reduction(max:err) tile(32,32)
for( int j = 1; j < n-1; j++) {
   for(int i = 1; i < m-1; i++) {</pre>
```



Create 32x32 tiles of the loops to better exploit data locality.

```
Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
A[j-1][i] + A[j+1][i]);
```

```
err = max(err, abs(Anew[j][i] - A[j][i]));
}
```

```
#pragma acc parallel loop tile(32,32)
for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
        A[j][i] = Anew[j][i];
        }
    }
    iter++;
OpenACC @</pre>
```

REBUILD THE CODE

pgcc -fast -ta=tesla -Minfo=accel laplace2d_uvm.c
main:

- 60, Generating copy(A[:m*n])
 Generating copyin(Anew[:m*n])
- 64, Accelerator kernel generated Generating Tesla code
 - 64, Generating reduction(max:error)
 - 65, #pragma acc loop gang /* blockIdx.x */
 - 67, #pragma acc loop vector(128) /* threadIdx.x */
- 67, Loop is parallelizable
- 75, Accelerator kernel generated

Generating Tesla code

- 76, #pragma acc loop gang /* blockIdx.x */
- 78, #pragma acc loop vector(128) /* threadIdx.x */
- 78, Loop is parallelizable



Now data movement only happens at our data region.



OPENACC SPEED-UP



LOOP OPTIMIZATION RULES OF THUMB

- It is rarely a good idea to set the number of gangs in your code, let the compiler decide.
- Most of the time you can effectively tune a loop nest by adjusting only the vector length.
- It is rare to use a worker loop. When the vector length is very short, a worker loop can increase the parallelism in your gang.
- When possible, the vector loop should step through your arrays
- Use the device_type clause to ensure that tuning for one architecture doesn't negatively affect other architectures.



OPENACC RESOURCES

Guides • Talks • Tutorials • Videos • Books • Spec • Code Samples • Teaching Materials • Events • Success Stories • Courses • Slack • Stack Overflow

Resources





Success Stories

https://www.openacc.org/success-stories



CLOSING REMARKS



KEY CONCEPTS

In this lab we discussed...

- How to profile a serial code to identify loops that should be accelerated
- How to use OpenACC's parallel loop directive to parallelize key loops
- How to use OpenACC's data clauses to control data movement
- How to optimize loops in the code for better performance


NEXT STEPS

Find more information at...

Please Connect with the Experts:

Tuesday & Wednesday 2-3, Thursday 11-12.

- Check your schedule for more OpenACC talks
- Network at the OpenACC Users Group Meeting, Tuesday 7:00PM @ Mosaic Restaurant (RSVP requested)
- Visit <u>https://www.openacc.org/events</u> for future opportunities

