Increasing the Performance of the NCAR Command Language via parallel processing

Christopher Kruse
Rick Brownrigg
NCL is an interpreted language designed specifically for scientific data analysis and visualization.

Portable, robust and free, NCL is available as binaries or open source.

Supports netCDF3/4, GRIB1/2, HDF-SDS, HDF4-EOS, binary, shapefiles, and ascii files.

Numerous analysis functions are built-in.

High quality graphics are easily created and customized with hundreds of graphic resources.

Many example scripts and their corresponding graphics are available.
NCAR Command Language

The NCAR Command Language (NCL) features array semantics.
The NCAR Command Language (NCL) features array semantics.

For example, instead of writing:

```plaintext
do = 1,N
   c(i) = a(i) * b(i)
end do
```

One can write:

```plaintext
c = a * b
```
The NCAR Command Language (NCL) features array semantics.

For example, instead of writing:

```
do = 1,N
  c(i) = a(i) * b(i)
end do
```

One can write:

```
c = a * b
```

With an array-based language and large datasets, there is a lot of opportunity for parallelism, but what should be parallelized and how should it be done?
How to Parallelize NCL?

In this project, commodity hardware was targeted that might be available to the average NCL user.
How to Parallelize NCL?

In this project, commodity hardware was targeted that might be available to the average NCL user.

OpenCL (Open Computing Language, OCL) was used to parallelize NCL
How to Parallelize NCL?

In this project, commodity hardware was targeted that might be available to the average NCL user.

OpenCL (Open Computing Language, OCL) was used to parallelize NCL.

OpenCL allows use of
- ATI/AMD GPUs
- Nvidia GPUs
- Multi-core CPUs
How to Parallelize NCL?

In this project, commodity hardware was targeted that might be available to the average NCL user.

OpenCL (Open Computing Language, OCL) was used to parallelize NCL

OpenCL allows use of
• ATI/AMD GPUs
• Nvidia GPUs
• Multi-core CPUs

OpenCL allows the largest amount of NCL users to take advantage of parallel processing
What to Parallelize?

Ideally, anything that requires looping with a large amount of iterations with a lot going on in the loops.
What to Parallelize?

Ideally, anything that requires looping with a large amount of iterations with a lot going on in the loops.

Areas of NCL targeted for parallelism in this project:
- Simple Operators ("+", ",", "+", etc.)
- Matrix Multiplication
- Transcendental Functions
  - "dim_avg_n" built-in function
  - "gc_inout" built-in function
What to Parallelize?

Ideally, anything that requires looping with a large amount of iterations with a lot going on in the loops.

Areas of NCL targeted for parallelism in this project:
Simple Operators ("+", ",", ",", "+", etc.)
Matrix Multiplication
Transcendental Functions
“dim_avg_n” built-in function
“gc_inout” built-in function

All kernels were written to be executed on either GPUs or CPUs.
Hardware Used

The parallelized NCL was tested on:

MacBook Pro Laptop
- Intel Dual Core CPU - 2.53 GHz
- Nvidia GeForce 9400M GPU - 1.10 GHz
  16 Cores

Linux (Ubuntu) Desktop
- Dual Quad Intel Xeon - 2.67 GHz
- Nvidia Quadro FX 3800 - 1.20 GHz
  192 Cores
Simple Operators

The simple operators were the first target of parallelization.
Simple Operators

The simple operators were the first target of parallelization.

Not an ideal problem for GPU programming, but parallelizing the simple ops was not too difficult.
The simple operators were the first target of parallelization.

Not an ideal problem for GPU programming, but parallelizing the simple ops was not too difficult.

While the number of iterations to do these operations can be large, not much is going on within the loops.
Simple Operators

The simple operators were the first target of parallelization.

Not an ideal problem for GPU programming, but parallelizing the simple ops was not too difficult.

While the number of iterations to do these operations can be large, not much is going on within the loops.

Very small performance increases were achieved only with unrealistically large arrays and a higher performance GPU.
Matrix Multiplication

Laptop:

Matrices larger than 64x64 and smaller than 1024x1024, NCL on the dual core CPU was fastest (~2x speed up).
Matrix Multiplication

Laptop:
Matrices larger than 64x64 and smaller than 1024x1024, NCL on the dual core CPU was fastest (~2x speed up).

Desktop:
NCL on the CPU and GPU was faster than native NCL for matrices larger than 128x128.
Matrix Multiplication

Laptop:
Matrices larger than 64x64 and smaller than 1024x1024, NCL on the dual core CPU was fastest (~2x speed up).

Desktop:
NCL on the CPU and GPU was faster than native NCL for matrices larger than 128x128.

The larger the matrix, the larger the performance increase using the GPUs.
Laptop:

NCL was slower using the CPUs and GPU, except when the GPU was used on an array with >$2^{28}$ elements.
Laptop:
NCL was slower using the CPUs and GPU, except when the GPU was used on an array with >2^28 elements.

Desktop:
NCL using the CPUs and GPU was faster for array sizes >2^16 elements.
Laptop:
NCL was slower using the CPUs and GPU, except when the GPU was used on an array with $>2^{28}$ elements.

Desktop:
NCL using the CPUs and GPU was faster for array sizes $>2^{16}$ elements

~4x speed ups on GPU for array sizes $>2^{21}$ elements
Built-in Function – dim_avg_n

Laptop:
Parallelized NCL was much slower for every size array
Laptop:
Parallelized NCL was much slower for every size array

Desktop:
Performance increases achieved only for arrays larger than 128x128x128
Built-in Function – `dim_avg_n`

**Laptop:**
Parallelized NCL was much slower for every size array

**Desktop:**
Performance increases achieved only for arrays larger than 128x128x128
Reduction was used for summing where needed
Built-in Function – gc_inout

Large performance increases for polygons with >10000 segments, except when the 9400m GPU was used.
Large performance increases for polygons with >10000 segments, except when the 9400m GPU was used.

Performance increases using every device for polygons of >=1000000 segments.
Built-in Function – gc_inout

Large performance increases for polygons with >10000 segments, except when the 9400m GPU was used.

Performance increases using every device for polygons of >=1000000 segments.

33x speed up using the Quadro FX 3800 against one core of the Dual Quad Xeon CPU.

Big win
Real World Example

The gc_inout function was used to determine which grid points of model output data were within boundaries of Pakistan.
The gc_inout function was used to determine which grid points of model output data were within boundaries of Pakistan.

84x80 coordinates were checked to see if they existed within a polygon of 102151 segments.
Real World Example

The gc_inout function was used to determine which grid points of model output data were within boundaries of Pakistan.

84x80 coordinates were checked to see if they existed within a polygon of 102151 segments.

<table>
<thead>
<tr>
<th>Software</th>
<th>Laptop Times</th>
<th>Desktop Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native NCL</td>
<td>7 min 38 sec</td>
<td>5 min 17 sec</td>
</tr>
<tr>
<td>OCLNCL-CPU Speed up</td>
<td>13.9x</td>
<td>3.6x</td>
</tr>
<tr>
<td>OCLNCL-GPU Speed up</td>
<td>2.2x</td>
<td>9.6x</td>
</tr>
</tbody>
</table>
Conclusions

It is possible to speed up NCL on commodity hardware available to the average NCL user.
Conclusions

It is possible to speed up NCL on commodity hardware available to the average NCL user.

On computers with lower performance GPUs, meaningful performance increases can be achieved using multi-core CPUs with applications that do not involve reduction.
Conclusions

It is possible to speed up NCL on commodity hardware available to the average NCL user.

On computers with lower performance GPUs, meaningful performance increases can be achieved using multi-core CPUs with applications that do not involve reduction.

On higher performance hardware, performance increases were achieved using NCL parallelized on both CPUs and a GPU, with the GPU being the fastest for the largest size problem in every application.
Future Work

Test the parallelized NCL interpreter on higher performance hardware.
Future Work

Test the parallelized NCL interpreter on higher performance hardware.

Parallelize the many other built-in functions in NCL.
Future Work

Test the parallelized NCL interpreter on higher performance hardware.

Parallelize the many other built-in functions in NCL.

Experiment with automatically generated kernels, which are compiled “just in time.”
Acknowledgements

Thanks to:

Rick Brownrigg - For mentoring me and helping me get familiar with how NCL works

Mohammad Abouali - For discussions of GPU programming

Thiyagarajan Chockalingam - For resources on optimizing parallel reduction on a GPU