Cross-Platform OpenCL Code and Performance Portability for CPU and GPU Architectures investigated with a Climate and Weather Physics Model

Han Dong
Dibyajyoti Ghosh
Fahad Zafar
Shujia Zhou

Motivation

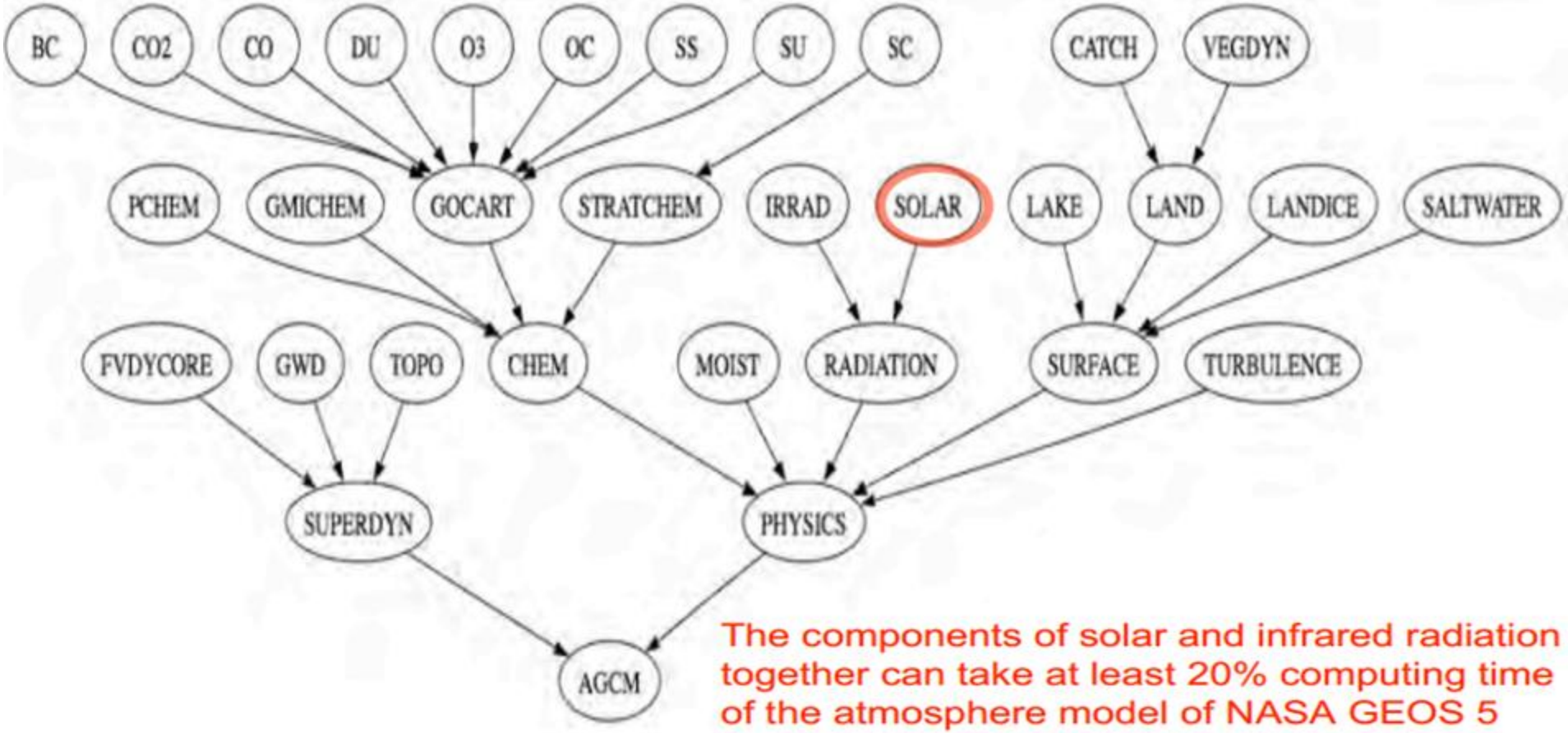
- Explore OpenCL in accelerating a real world computationally intensively application. (NASA climate and weather physics model)
- Investigate both the performance and code portability of OpenCL with GPUs and CPUs.
- Extend the work of Zafar et al [1] by:
  - Producing a baseline OpenCL code that compiles and runs on both CPUs and GPUs.
  - Maintain the accuracy of serial code.

Outline

- Solar Radiation Model
- Experimental Setup
- Porting and Optimizations
- Results
- Explicit AVX Registers
- Conclusion

SOLAR RADIATION MODEL

NASA GEOS-5 Code Structure

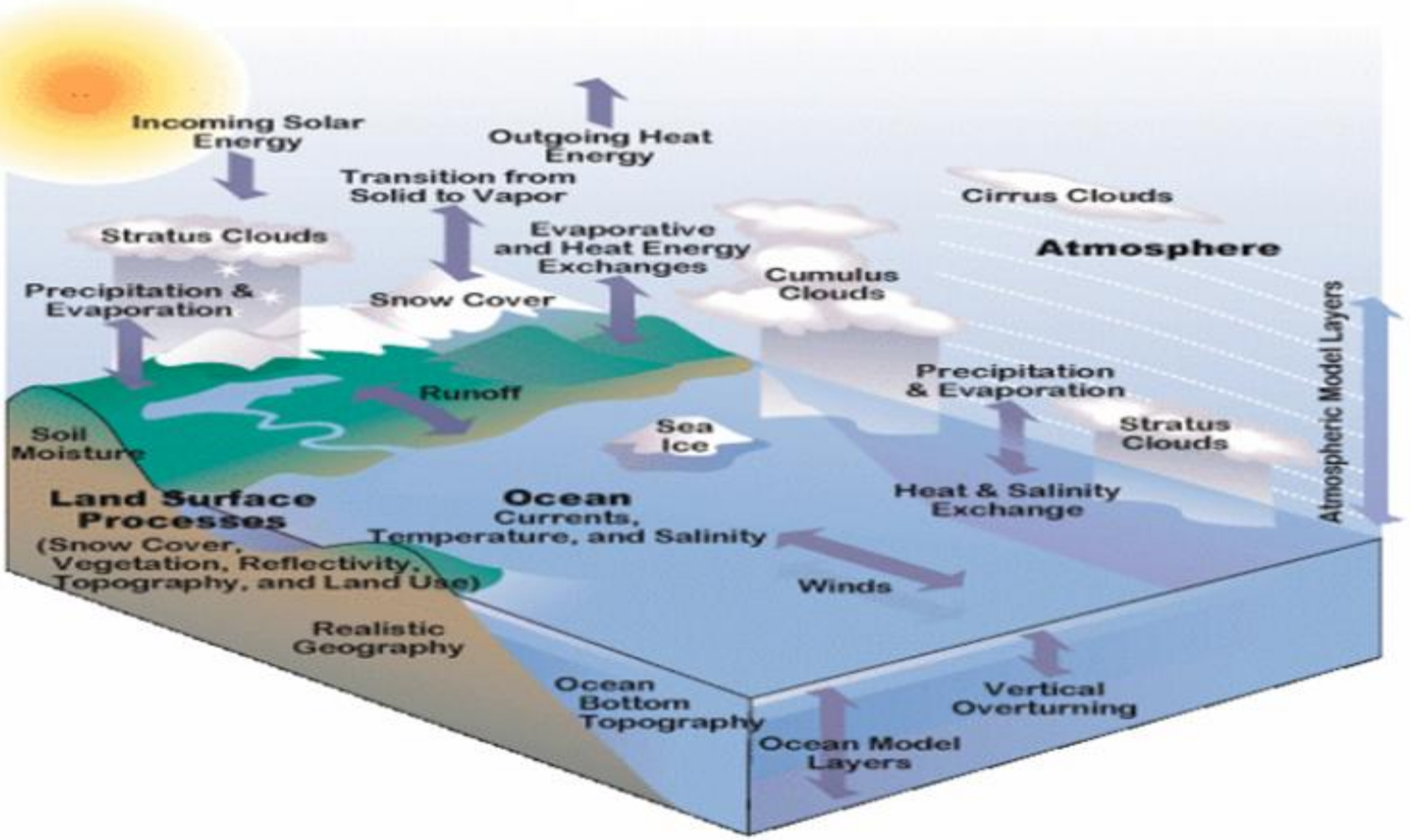


The components of solar and infrared radiation together can take at least 20% computing time of the atmosphere model of NASA GEOS 5

NASA GEOS-5

- Solar radiation component of NASA's GEOS-5 takes **~10%** of model computation time.
- NASA is interested in analysis of performance and cost benefit using non traditional computing systems.
- GEOS-5 - **20+** years old, written in Fortran (mostly), still evolving.
- Cannot be entirely rewritten due to production constraints.

Processes in a Climate Model



http://www.ucar.edu/communications/CCSM/overview.html

Code Structure of SOLAR

SORAD
- (data structure initializations)

SOLUV
- (write initial data to arrays)
- computeCCandKK
- Cldscale
- cldscaleToBandLoop
- FOR loop to integrate over spectral bands
  - *BandLoopStartToDeledd*
  - *deledd*
  - *cldflx*
  - *(finalize data)*

SOLIR
- (write initial data into arrays)
- FOR loop to integrate over spectral bands
  - *bandLoopStartToCldscale*
  - *computeCCandKK*
  - *cldscale*
  - FOR loop to calculate water vapor absorption for 10 k intervals
    - *kLoopStartToDeledd*
    - *deledd*
    - *cldflx*
    - *(finalize data)*

- (Finalize data and output results)

Experimental Setup

Platform	Compute Units	Clock (GHz)	Environment	GCC Version	OpenCL SDK	OpenCL Specification
IBM JS22 Power6	8	4.00	Red Hat 4.1.2-48	4.1.2	IBM Power v0.3	1.1
Intel Core i7 2630QM	8	2.00	Ubuntu 11.04	4.5.2	Intel 1.5	1.1
Intel Core 2 Duo P8600	2	2.40	Mac OSX 10.6.7	4.2.1	Intel 1.1	1.0
GeForce GTX 580M	8	-	Ubuntu 11.04	4.5.2	CUDA 4.0.1	1.1
GeForce GT 320M	6	-	Mac OSX 10.6.7	4.2.1	CUDA 3.2	1.0
Dual Intel Xeon X5670	24	2.93	Red Hat 4.4.4-13	4.4.4	Intel 1.5	1.1

TABLE I

CHARACTERISTICS OF CPU'S AND GPU'S USED IN PERFORMANCE TESTING.

PORTING AND OPTIMIZATIONS

OpenCL Compilation Model

- OpenCL uses Dynamic/Runtime compilation model [2]
  1. Code is first compiled to an Intermediate Representation (IR)
     - Done once and IR is stored
  2. IR is compiled to machine code for execution
     - Application loads IR and performs compilation during run time
- Preprocessor macros were used for constant variables that dictated kernel loop iterations.
- Preprocessor macros enable OpenCL dynamic compilation to ensure that the variable is known at kernel compile time allowing compilers to perform implicit loop unrolling.

CLDFLX Serial

```
for (ih=0; ih<2; ih++)
    for (mb=0; mb<M_BLOCK; mb++)
        //calculate tda[0][ih][0][mb], tta[...], rsa[...]
        //calcualte tda[1][ih][0][mb], tta[...], rsa[...]
        //calculate rra[ih][0][LM+1][mb], rxa[...]
        //calculate rra[ih][1][LM+1][mb], rxa[...]
    for (l=1; l<ict; l++)
        for (mb=0; mb<M_BLOCK; mb++)
            //calculate tda[0][ih][l][mb], tta[...], rsa[...]
            //calculate tda[0][ih][l][mb], tta[...], rsa[...]
    for (im=im1-1; im<im2; im++)
        for (l=ict; l<icb; l++)
            for (mb=0; mb<M_BLOCK; mb++)
                //update tda[m][ih][l].., tta[m][ih][l].., rsa[m][ih][l]..
for (ih=0; ih<2; ih++){
    if(ih==0) .. for (mb=0; mb<M_BLOCK; mb++) { //clear portion, update ch[mb] }
    else .. for (mb=0; mb<M_BLOCK; mb++) { //cloudy portion, update ch[mb] }
    for (im=0; im<2; im++){
        if(im==0) .. for (mb=0; mb<M_BLOCK; mb++) { //clear portion, update cm[mb] }
        else .. for (mb=0; mb<M_BLOCK; mb++) { //cloudy poriton, update cm[mb] }
        for (is=0; is<2; is++) {
            if(is==0) .. for (mb=0; mb<M_BLOCK; mb++) { //clear portion, update ct[mb] }
            else for (mb=0; mb<M_BLOCK; mb++) { //cloudy portion, update ct[mb] }
        for (l=icb; l<LM+1; l++)
            for (mb=0; mb<M_BLOCK; mb++)
                //update tda[im][ih][l][mb], tta[....], rsa[....]
        for (l=ict; l>-1; l--)
            for (mb=0; mb<M_BLOCK; mb++)
                //update rra[is][im][l][mb], rxa[....]
        for (l=1; l<=LM+1; l++)
            for (mb=0; mb<M_BLOCK; mb++)
                //update fdndif[mb], flxdn[l][mb]
        for (l=0; l<LM+1; l++)
            for (mb=0; mb<M_BLOCK; mb++)
                //update fall[l][mb], fsdir[mb], fsdif[mb]
```

Initialize

Update

Finalize

CLDFLX Parallel

DownKernel

```
int mask1[8] = {0,1,0,1,0,1,0,1};
int mask2[8] = {0,0,1,1,0,0,1,1};
int mask3[8] = {0,0,0,0,1,1,1,1};
i = get_global_id(0); j = get_global_id(1);
ih = mask1[j]; im = mask2[j]; is = mask3[j];
k = j * 16 + i;
stride3D = ih*(NLM+2)*NM_BLOCK;
stride5D = is*2*(NLM+2)*NM_BLOCK+im*(NLM+2)*NM_BLOCK;
temp = rr[is * (NLM+2) * NM_BLOCK + (NLM+1) * NM_BLOCK + k];
rra[is * 2 * (NLM+2) * NM_BLOCK + im * (NLM+2) * NM_BLOCK + (NLM+1) * NM_BLOCK + k] = temp;
rxa[is * 2 * (NLM+2) * NM_BLOCK + im * (NLM+2) * NM_BLOCK + (NLM+1) * NM_BLOCK + k] = temp;
dev_rra0 = temp;
dev_rxa0 = temp;
for (l = NLM; l >= 0; l --)
    temp_rr = rr[stride3D+l*NM_BLOCK+k];
    temp_rs = rs[stride3D+l*NM_BLOCK+k];
    temp_td = td[stride3D+l*NM_BLOCK+k];
    temp_ts = ts[stride3D+l*NM_BLOCK+k];
    temp_tt = tt[stride3D+l*NM_BLOCK+k];
    denm = temp_ts / (1.0 - temp_rs * dev_rxa0);
    dev_rra1 = temp_rr + (temp_td * dev_rra0 + (temp_tt - temp_td) * dev_rxa0) * denm;
    dev_rxa1 = temp_rs + temp_ts * dev_rxa0 * denm;
    rra[stride5D+l*NM_BLOCK+k] = dev_rra1;
    rxa[stride5D+l*NM_BLOCK+k] = dev_rxa1;
    dev_rra0 = dev_rra1;
    dev_rxa0 = dev_rxa1;
```

CLDFLX Parallel

```
for(l = 0; l <= NLM + 1; l++)
    temp_dev_rxa = rxa[stride5D + l * NM_BLOCK + k];
    temp_dev_rra = rra[stride5D + l * NM_BLOCK + k];
    denm = 1.0 / (1.0 - dev_rsa0 * temp_dev_rxa);
    xx = dev_tda0 * temp_dev_rra;
    yy = dev_tta0 - dev_tda0;
    temp_fdndif = (xx * dev_rsa0 + yy) * denm;
    fupdif = (xx + yy * temp_dev_rxa) * denm;
    flxdn[l * NM_BLOCK + k] = dev_tda0 + temp_fdndif - fupdif;
    if(l == (NLM+1))
        fdndir[k] = dev_tda0;
        fdndif[k] = temp_fdndif;
    if(l < (NLM+1))
        temp_rr = rr[stride3D+l*NM_BLOCK+k];
        temp_rs = rs[stride3D+l*NM_BLOCK+k];
        temp_td = td[stride3D+l*NM_BLOCK+k];
        temp_ts = ts[stride3D+l*NM_BLOCK+k];
        temp_tt = tt[stride3D+l*NM_BLOCK+k];
        denm = temp_ts / (1.0 - dev_rsa0 * temp_rs);
        dev_tda1 = dev_tda0 * temp_td;
        dev_tta1 = dev_tda0 * temp_tt + (dev_tda0 * dev_rsa0 * temp_rr + dev_tta0 - dev_tda0) * denm;
        dev_rsa1 = temp_rs + temp_ts * dev_rsa0 * denm;
        dev_tda0 = dev_tda1;
        dev_tta0 = dev_tta1;
        dev_rsa0 = dev_rsa1;
```

UpKernel

CLDFLX Parallel

```
int mask1[8] = {0,1,0,1,0,1,0,1};
int mask2[8] = {0,0,1,1,0,0,1,1};
int mask3[8] = {0,0,0,0,1,1,1,1};
mb = get_global_id(0);
l = get_global_id(1);
k = get_global_id(2);
ih = mask1[k]; im = mask2[k]; is = mask3[k];
fclr[l * NM_BLOCK + mb] = flxdn[(l+1) * (NM_BLOCK+1) + mb];
fall[l * NM_BLOCK + mb] = fall[l * NM_BLOCK + mb] + flxdn[(l+1) * (NM_BLOCK+1) + mb] * ct_data[(ih * 4 + im * 2 + is * 1) * 128 + mb];
```

ReductionKernel

RESULTS

Accomplishments

- A single parallel OpenCL code runnable across multiple platforms consisting of IBM Cell Processors, multicore CPUs and GPUs.

- Achieved parallel implementation accuracy of 1.0×10^{-6} in numerical differences when compared to serial implementation (increased from 1.0×10^{-4} of Fahad et al [1]).

- Discovered OpenCL can enable CPU devices to achieve dramatic performance improvements.

Performance Results

	Column Size							
	128		256		512		1024	
Platform	**Per Thread**	**Total**	**Per Thread**	**Total**	**Per Thread**	**Total**	**Per Thread**	**Total**
Intel Core i7-2630QM	5.5	55	11.25	90	24	192	17.7	142
Dual Intel Xeon X5670	3.3	80	6.8	163.7	14.9	359	13	312
GeForce GTX 580M	-	21	-	38	-	61.5	-	26
IBM JS22 Power6	1.3	10.2	0.9	7.16	0.27	2.16	0.742	5.93
Intel Core 2 Duo	1.01	2.02	1.05	2.1	0.89	0.445	-	-
GeForce GT 320M	-	10.02	-	5.329	-	-	-	-

TABLE II

SPEEDUP ACROSS ALL PLATFORMS

Assembly Dump

```
vmulpd YMM1, YMM4, YMM1
vpshufd XMM4, XMM5, 3
vcvtss2sd XMM4, XMM4, XMM4
vmovhlps XMM8, XMM5, XMM5
vcvtss2sd XMM8, XMM8, XMM8
```

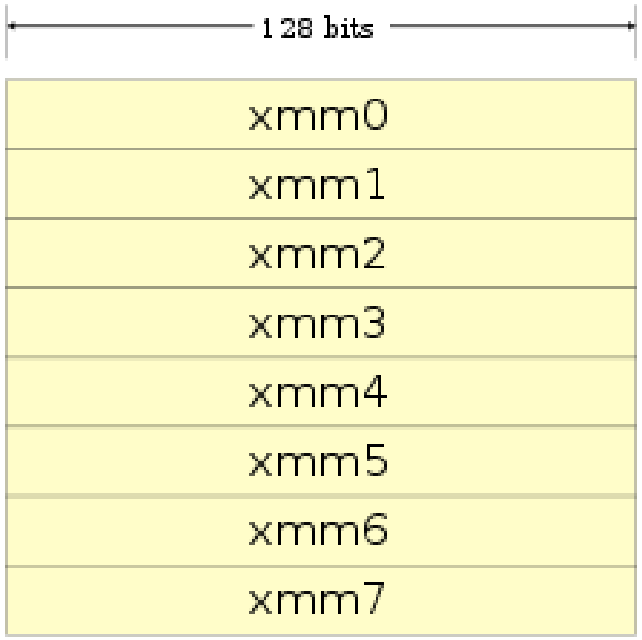
Listing 1. OpenCL offline compiler assembly dump of a portion of kernel code on Intel i7-2630QM.

```
vmulss XMM0, XMM0, DWORD PTR [RSP +
    84]
vmovss XMM1, DWORD PTR [RIP + .
    LCPI56_0]
vaddss XMM2, XMM0, XMM1
vmovss DWORD PTR [RSP + 60], XMM2
```

Listing 2. OpenCL offline compiler assembly dumping of a portion of kernel code on Intel Xeon X5670.

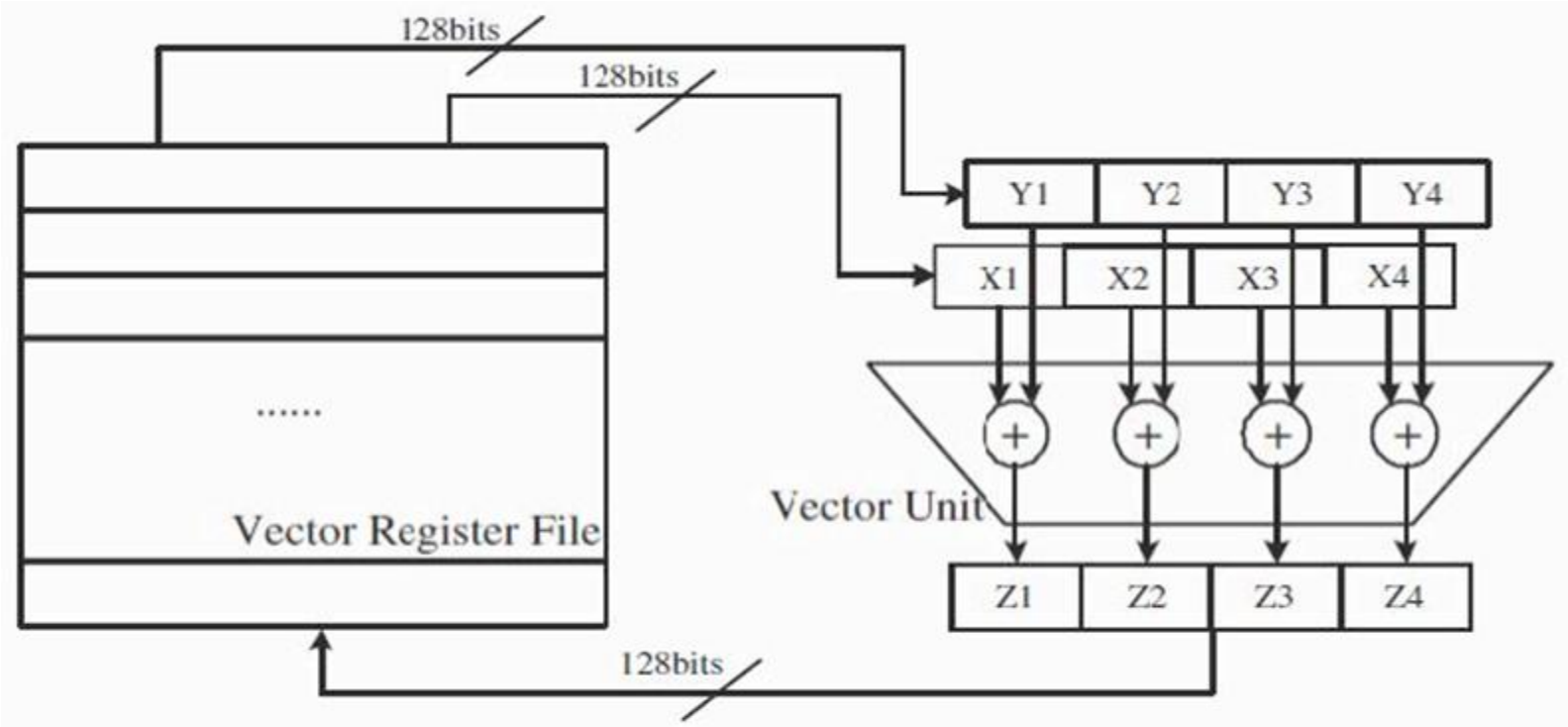
Intel Streaming SIMD Extensions

- Designed by Intel and introduced in 1999.
- Increases performance when the same operation are performed on multiple data objects.
- Registers:
  - SSE
  - SSE2
  - SSE3
  - SSE4
  - AVX



How does it work?

- Intel SSE packs multiple data into fixed size registers and applies same instructions to all data in parallel.



How does OpenCL contribute?

- OpenCL coding style is SIMD based as it is intended to run on GPUs.
- Optimizations that are important for GPUs such as reducing thread divergence and improving coalesced memory accesses helps CPU compilers.
- SIMD style of kernel programming eliminates complex loop constructs. This helps compilers by providing more effective vectorization as it usually behaves in a conservative manner for vectorization [3][4].
- Data dependence and cycles are broken through the optimization of kernels originally intended to execute on GPUs to fully exploit the SIMD feature of CPU vector processors.

GPU Results

- Reduced the original 70 kernels from Zafar et al [1] to about half (36 kernels).
- Exploring local memory was severely limited due to the simplified kernels.
- Development Time vs Performance

Explicit AVX Registers

- Difficulties:
  - Affect the performance portability due to targeting a specific vector width
  - Vector data types cannot be used in conditional statement
    - Utilized built-in relational functions such as ***isgreater*** or ***isless*** and called stub functions for each side of the conditional
  - Pad arrays to be divisible by 8

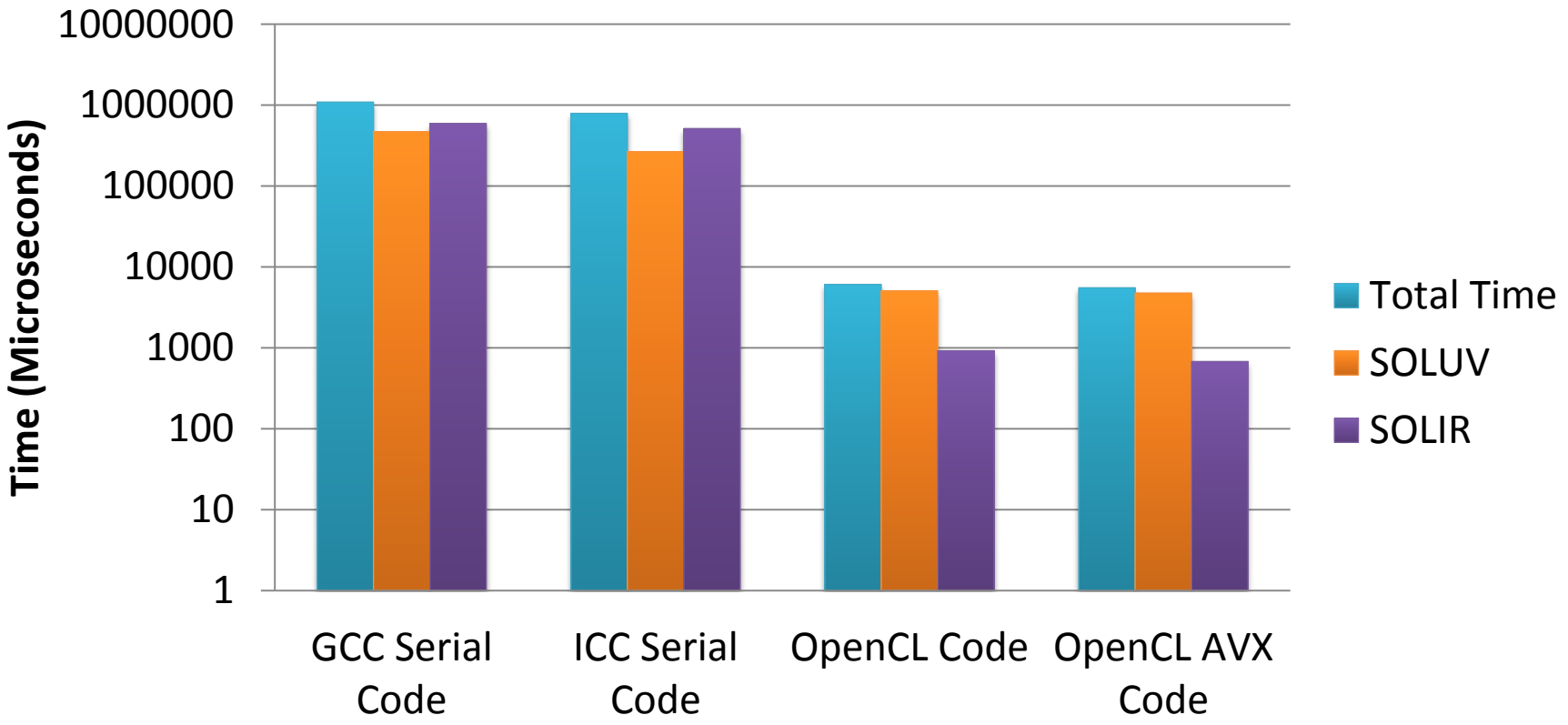
```
vmovaps  YMM0, YMMWORD PTR [RIP + .
   LCPI16_0]
vdivps       YMM0, YMM0, YMMWORD PTR [
   R12 + R13]
mov      RAX, QWORD PTR [RBP + 24]
vmovaps  YMMWORD PTR [RAX + R13], YMM0
mov      RAX, QWORD PTR [RBP + 32]
vmovaps  YMM0, YMMWORD PTR [RAX + R13]
vmaxps       YMM0, YMM0, YMMWORD PTR [
   RIP + .LCPI16_1]
vmovaps  YMMWORD PTR [RSP], YMM0 # 32-
   byte Spill
```

Listing 4. Intel OpenCL Offline Compiler output of assembly on Intel Core i7 with explicit AVX.

```
vmovq XMM0, RAX
vmovlhps XMM0, XMM0, XMM0
vmovaps  XMMWORD PTR [RSP + 192], XMM0
vmovaps  YMM1, YMMWORD PTR [RIP + .
   LCPI9_0]
vextractf128   XMM2, YMM1, 1
vpaddq       XMM2, XMM0, XMM2
vpshufd  XMM2, XMM2, 8
vmovaps  YMMWORD PTR [RSP + 160], YMM1
```

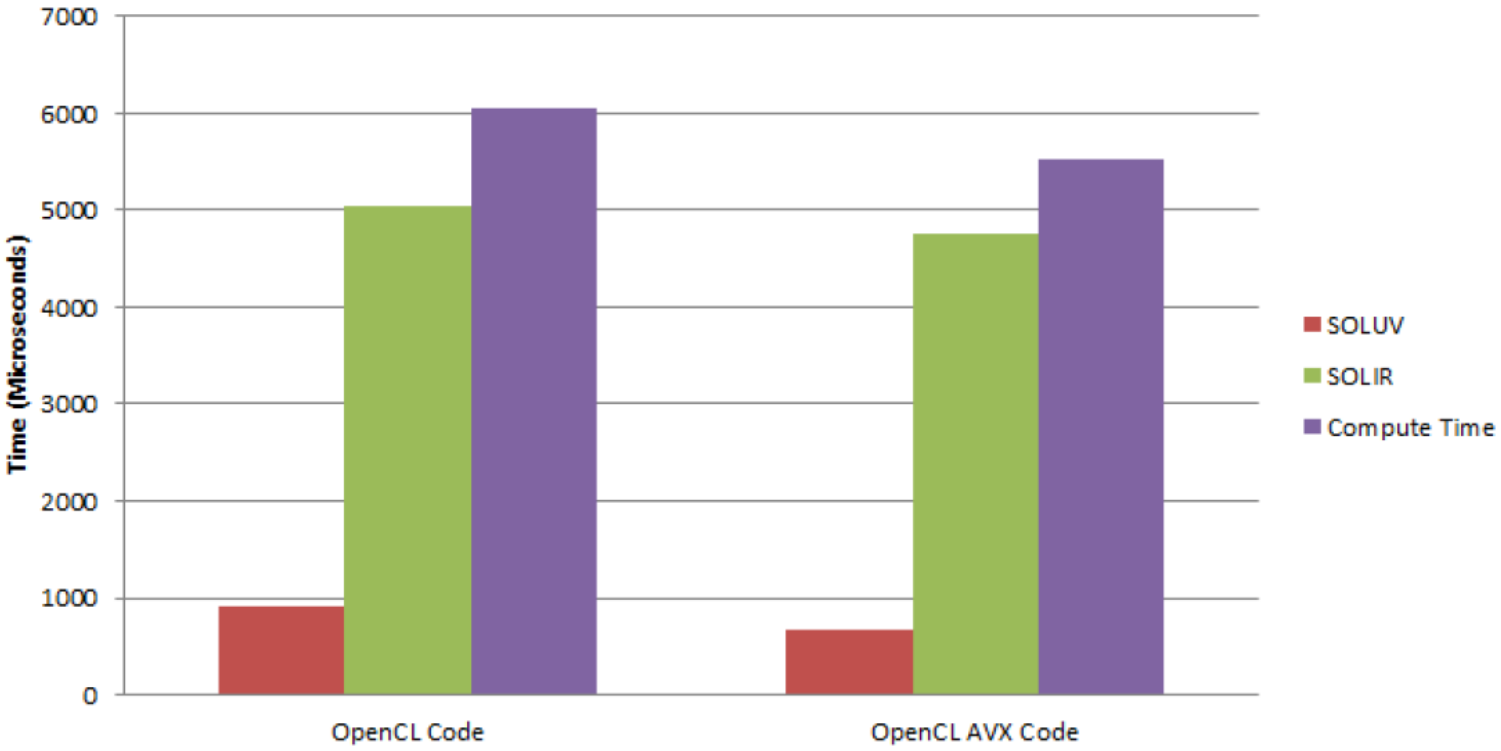
Listing 5. Intel OpenCL Offline Compiler output of assembly on Intel Core i7 WITHOUT explicit AVX.

Intel ICC Compiler Comparisons



Execution time comparisons of serial code compiled with GCC, serial code compiled with Intel ICC (12.1.4) on Intel i7-2630QM CPU, and parallel OpenCL implementations.

Performance Results



Execution time comparison between OpenCL code and OpenCL code using explicit AVX intrinsic on Intel Core i7-2630QM CPU on 128 column size.

Conclusion

- Developed an OpenCL code for a representative climate and weather physics model that is able to run across multiple platforms.

- OpenCL's kernel programming and execution model facilitates the compiler to vectorize the code and consequently improve performance.

References

[1] F. Zafar, D. Ghosh, L. Sebald, and S. Zhou, “Accelerating a climate physics model with OpenCL,”Symposium on Application Accelerators in High-Performance Computing 2011, 2011.

[2] Intel, “Writing optimal opencl code with intel opencl sdk,” http://software.intel.com/file/39189, 2011.

[3] M. Garzarn and S. Maleki, “Program optimization through loop vectorization,” http://agora.cs.illinois.edu/download/attachments/38305904/9-Vectorization.pdf, 2010.

[4] C. M. J. Garzaran, “Loop vectorization,” https://agora.cs.illinois.edu/download/attachments/28937737/10-Vectorization.pdf, 2010.