Tuned OpenMP to CUDA Translation

Rudi Eigenmann
Seyong Lee
Purdue University
Motivation

- Many-Core Processors: New Opportunities for General-Purpose High-Performance Computing
  - Multicore CPUs
  - Cell BE
  - GPGPUs
  - Multicore FPGAs
  - Etc.
Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU

(a) Peak GFLOP/s

(b) Peak Memory Bandwidth

Courtesy: NVIDIA
Tradeoff between Performance and Programmability in Many-Core Processors

- GPU provides more computing power but worse programmability than CPU.
- GPU architectures are optimized for stream computations.
- General-Purpose GPUs (GPGPUs) provide better programmability for general applications.
  - CUDA programming model is more user-friendly than previous approaches, but still complex and error-prone.
Goal and Approach

- Improve **programmability** and achieve high **performance** in the new many-core architectures such as GPGPUs.

**Approach**

- Use OpenMP for easier programming on CUDA-based GPGPUs.
- Propose compile-time optimization techniques.
- Extend OpenMP to allow fine-grained control of CUDA-related parameters and optimizations.
Contribution

- Propose a new API, called OpenMPC, for improved CUDA programming, which provides both programmability and tunability.
  - Developed a fully automatic and parameterized reference compilation system to support OpenMPC.
  - Developed several tools to assist users in performance tuning.

- Evaluation on the 14 OpenMP programs:
  - User-assisted tuning with optional manual modification of input OpenMP programs improves performance
  - **1.24 times** on average (up to **7.71 times**) over un-tuned versions, which is
  - **75%** (92% if excluding one benchmark) of the performance of hand-written CUDA versions.
Related Work

GPU Code Generation

- Manual
  - CUDA [07]
  - OpenCL [08]
  - BrookGPU [04]
  - Ryoo [08]
- Semi-Automatic
  - CUDA-lite [08]
  - JCUDA [09]
- Automatic
  - G-ADAPT [09]
  - Nukada [09]
  - OpenMP-to-GPGPU [09]
  - hiCUDA [09]
  - PGI Accelerator [09]
  - OpenMPC [10]
  - OpenMPC-to-GPGPU [09]

Manual

Semi-Automatic

Automatic

- Baskaran [08]
- Volkov [08]
- Baskaran [10]
- Leung [10]
Overview of the CUDA Programming Model

- The CUDA Programming Model
  - A general-purpose multi-threaded SIMD model for GPGPU programming
  - A CUDA program consists of a series of sequential and parallel execution phases.
    - Parallel regions are executed by a set of threads (thread batching)

- Architectural Limit
  - No global synchronization mechanism is supported.
  - No H/W caching mechanism for global memory

CUDA Programming Model

Courtesy: NVIDIA
A host CPU and a GPU device have separate address space.
The shared memory and register bank in a multiprocessor are dynamically partitioned among the active thread blocks running on the multiprocessor.
Programming Complexity Comparison (Jacobi)

OpenMP code

```c
float a[SIZE_2][SIZE_2];
float b[SIZE_2][SIZE_2];

int main (int argc, char * argv[]) {
    int i, j, k;
    for (k = 0; k < ITER; k++) {
        #pragma omp parallel for private(i, j)
        for (i = 1; i <= SIZE; i++)
            for (j = 1; j <= SIZE; j++)
                a[i][j] = (b[i - 1][j] + b[i + 1][j] + b[i][j - 1] + b[i][j + 1]) / 4.0f;
        #pragma omp parallel for private(i, j)
        for (i = 1; i <= SIZE; i++)
            for (j = 1; j <= SIZE; j++)
                b[i][j] = a[i][j];
    }
    return 0;
}
```

CUDA code

```c
float a[SIZE_2][SIZE_2];
float b[SIZE_2][SIZE_2];
__global__ void main_kernel0(float a[SIZE_2][SIZE_2], float b[SIZE_2][SIZE_2]) {
    int i, j;
    int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
    int _gtid = (threadIdx.x+(_bid*blockDim.x));
    i=(_gtid+1);
    if (i<=SIZE { for (j=1; j<=SIZE j ++ )
        a[i][j] = (b[(i-1)][j]+b[(i+1)][j]+b[i][(j-1)]+b[i][(j+1)])/4.0F;
    } }

__global__ void main_kernel1(float a[SIZE_2][SIZE_2], float b[SIZE_2][SIZE_2]) {
    int i, j;
    int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
    int _gtid = (threadIdx.x+(_bid*blockDim.x));
    i=(_gtid+1);
    if (i<=SIZE { for (j=1; j<=SIZE j ++ )
        b[i][j] = a[i][j];
    } }

int main(int argc, char * argv[]) {
    int i, j, k;
    float * gpu__a; float * gpu__b;
    gpuBytes=(SIZE_2*SIZE_2)*sizeof (float);
    cudaMalloc((void * )( & gpu__a), gpuBytes);
    dim3 dimBlock0(BLOCK_SIZE, 1, 1);
    dim3 dimGrid0(NUM_BLOCKS, 1, 1);
    cudaMemcpy(gpu__a, a, gpuBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b, gpu__b, gpuBytes, cudaMemcpyDeviceToHost);
    for (k=0; k<ITER; k++) {
        main_kernel0<<<dimGrid0, dimBlock0, 0, 0>>>((float *)gpu__a, (float *)gpu__b);
        main_kernel1<<<dimGrid1, dimBlock1, 0, 0>>>((float *)gpu__a, (float *)gpu__b);
        cudaMemcpy((void * )(void * )gpu__a, (void * )(void * )gpu__b), cudaMemcpyDeviceToHost);
    }
    cudaMemcpy(gpu__a, (void * )(void * )gpu__a, (void * )(void * )gpu__b), cudaMemcpyDeviceToHost);
    return 0;
```
OpenMP Version of Jacobi

```c
float a[SIZE_2][SIZE_2];
float b[SIZE_2][SIZE_2];
int main (int argc, char *argv[]) {
    int i, j, k;
    for (k = 0; k < ITER; k++) {
        #pragma omp parallel for private(i, j)
        for (i = 1; i <= SIZE; i++)
            for (j = 1; j <= SIZE; j++)
                a[i][j] = (b[i - 1][j] + b[i + 1][j] + b[i][j - 1] + b[i][j + 1]) / 4.0f;
        #pragma omp parallel for private(i, j)
        for (i = 1; i <= SIZE; i++)
            for (j = 1; j <= SIZE; j++)
                b[i][j] = a[i][j];
    return 0;
}
```
int main(int argc, char * argv[]) {
    int i, j, k;
    float * gpu_a; float * gpu_b;
    gpuBytes=(SIZE_2*SIZE_2)*sizeof(float);
        CUDA_SAFE_CALL(cudaMalloc(((void **)( & gpu_a)), gpuBytes));
    CUDA_SAFE_CALL(cudaMemcpy(gpu_a, a, gpuBytes, cudaMemcpyHostToDevice));
    gpuBytes=((SIZE_2*SIZE_2)*sizeof(float));
    CUDA_SAFE_CALL(cudaMalloc(((void **)( & gpu_b)), gpuBytes));
    CUDA_SAFE_CALL(cudaMemcpy(gpu_b, b, gpuBytes, cudaMemcpyHostToDevice));
    dim3 dimBlock0(BLOCK_SIZE, 1, 1);
    dim3 dimGrid0(NUM_BLOCKS, 1, 1);
    dim3 dimBlock1(BLOCK_SIZE, 1, 1);
    dim3 dimGrid1(NUM_BLOCKS, 1, 1);
    for (k=0; k<ITER; k++) {
        main_kernel0<<<dimGrid0, dimBlock0, 0, 0>>>(((float *)[SIZE_2])gpu_a,
            (float *)[SIZE_2])gpu_b);
        main_kernel1<<<dimGrid1, dimBlock1, 0, 0>>>(((float *)[SIZE_2])gpu_a,
            (float *)[SIZE_2])gpu_b);
    }
    gpuBytes=(SIZE_2*SIZE_2)*sizeof(float);
    CUDA_SAFE_CALL(cudaMemcpy(b, gpu_b, gpuBytes, cudaMemcpyDeviceToHost));
    gpuBytes=(SIZE_2*SIZE_2)*sizeof(float);
    CUDA_SAFE_CALL(cudaMemcpy(a, gpu_a, gpuBytes, cudaMemcpyDeviceToHost));
    CUDA_SAFE_CALL(cudaFree(gpu_b));
    CUDA_SAFE_CALL(cudaFree(gpu_a));
    fflush(stdout); fflush(stderr);
    return 0;
}
CUDA Version of Jacobi (2)

```c
__global__ void main_kernel0(float a[SIZE_2][SIZE_2], float b[SIZE_2][SIZE_2]) {
    int i,j;
    int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
    int _gtid = (threadIdx.x+(_bid*blockDim.x));
    i=(_gtid+1);
    if (i<=SIZE {
        for (j=1; j<=SIZE j ++ )
            a[i][j]=(b[(i-1)][j]+b[(i+1)][j]+b[i][(j-1)]+b[i][(j+1)])/4.0F;
    }
}

__global__ void main_kernel1(float a[SIZE_2][SIZE_2], float b[SIZE_2][SIZE_2]) {
    int i,j;
    int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
    int _gtid = (threadIdx.x+(_bid*blockDim.x));
    i=(_gtid+1);
    if (i<=SIZE {
        for (j=1; j<=SIZE j ++ )
            b[i][j]=a[i][j];
    }
}
```
Why OpenMP?

- Advantages of OpenMP as a programming paradigm for GPGPUs.
  - Loop-level parallelism of OpenMP is an ideal target for utilizing GPU’s highly parallel computing units.
  - OpenMP’s fork-join model well represents the relationship between a master thread in CPU and a set of worker threads in a GPU.
  - Incremental parallelization of OpenMP can add the same benefit to GPGPU programming.
Baseline Translation of OpenMP into CUDA

- Identify Kernel Regions (parallel work to be executed on the GPU)

1) Split an OpenMP parallel region at every synchronization point.

2) Select sub-parallel regions containing at least one omp-for loop and transform them into kernel functions.
Unoptimized Performance of OpenMP Programs on CUDA

Speedups are over serial on the CPU, when the largest available input data were used.

Experimental Platform: CPU: two Dual-Core AMD Opteron at 3 GHz
GPU: NVIDIA Quadro FX 5600 with 16 multiprocessors at 1.35GHz
Differences between OpenMP and CUDA Programming Models

**OpenMP Model**
- Coarse-grain parallelism (runs tens of threads.)
- SIMD/MIMD model
- Threads work well on both regular and irregular memory access patterns.
- Optimized for traditional shared-memory multiprocessors (SMPs)

**CUDA Model**
- Fine-grain parallelism (runs thousands of threads.)
- SIMD model
- Threads strongly prefer regular memory access patterns
- Optimized for *stream architectures* that operate on a large data space (or stream) in parallel and tuned for fast access to regular, consecutive data.
Intra-Thread vs. Inter-Thread Locality

- Intra-thread locality is beneficial to both OpenMP and CUDA model.
- Inter-thread locality plays a critical role in CUDA model.
Compiler Optimizations

- Techniques to Optimize GPU Global Memory Accesses
  - Parallel Loop Swap
  - Loop Collapsing
  - Matrix Transpose
- Techniques to Exploit GPU On-chip Memories
- Techniques to Optimize Data Movement between CPU and GPU
  - Resident GPU Variable Analysis
  - Live CPU Variable Analysis
  - Memory Transfer Promotion Optimization
Optimized Performance of OpenMP Programs on CUDA

- Speedups are over serial on the CPU, when the largest available input data were used.
- **Experimental Platform:**
  - CPU: two Dual-Core AMD Opteron at 3 GHz
  - GPU: NVIDIA Quadro FX 5600 with 16 multiprocessors at 1.35GHz
Why Developing Efficient CUDA Programs Still Remains Difficult?

- Static performance prediction of optimizations is difficult due to:
  - Complex interactions among hardware resources
  - Multi-layered software execution stack

OpenMP is a high abstraction
- Reduced control over fine-grained tuning
OpenMPC (OpenMP extended for CUDA)

- OpenMPC = OpenMP + a new set of directives and environment variables for CUDA.
- OpenMPC provides:
  - high level abstraction of the CUDA programming model (Programmability)
  - tuning environment to generate CUDA programs in many optimization variants (Tunability)
OpenMPC: Directive Extension and Environment Variables

- **OpenMPC Directive Format**
  
  ```
  #pragma cuda gpurun [clause [,] clause]…
  #pragma cuda cpurun [clause [,] clause]…
  #pragma cuda nogpurun
  #pragma cuda ainfo procname(pname) kernelid(kID)
  ```

- **OpenMPC Environment Variables**
  
  Control the program-level behavior of various optimizations or execution configurations for an output CUDA program.
For automatic tuning, additional passes are invoked between CUDA Optimizer and O2G Translator, marked as (A) in the figure.

```c
#pragma omp parallel shared(firstcol, lastcol, x, z) private(j)
reduction(+: norm_temp11, norm_temp12)
#pragma cuda ainfo kernelid(1) procname(main)
#pragma cuda gpurun noc2gmemtr(x, z)
#pragma cuda gpurun nocudamalloc(x, z)
#pragma cuda gpurun nocudafree(firstcol, lastcol, x, z)
#pragma cuda gpurun nog2cmemtr(firstcol, lastcol, x, z)
#pragma cuda gpurun sharedRO(firstcol, lastcol)
#pragma cuda gpurun texture(z)
{
    #pragma omp for private(j) nowait
    for (j=1; j<=((lastcol-firstcol)+1); j ++ ) {
        norm_temp11=(norm_temp11+(x[j]*z[j]));
        norm_temp12=(norm_temp12+(z[j]*z[j]));
    }
}
```
OpenMPC Tuning Framework

Exhaustive search was used in the prototype tuning system.
Evaluation

- Conducted Performance Tuning Experiments
  - Profile-based Tuning (fully automatic)
    - Find the best optimization configuration for a target program with a training input data set
    - The best variant is used to measure the performance with the actual data sets.
  - User-Assisted Tuning
    - The programs have been tuned for each production data set. (upper performance bound test)
    - Unsafe, aggressive optimizations are applied under the user’s approval. (All automatic except for user’s approval)
Overall Tuning Performance

- **Performance Summary**

<table>
<thead>
<tr>
<th>Translator Input</th>
<th>Performance Improvement over All-Opt Versions</th>
<th>Relative Performance over Manual Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIN</td>
<td>MAX</td>
</tr>
<tr>
<td>Orig. OpenMP</td>
<td>1</td>
<td>4.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mod. OpenMP</td>
<td>1</td>
<td>7.71</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In A(B) format, B refers the performance when the results of LUD are excluded.

- **Optimization Search Space Reduction**
  - **98.7%** on average for program-level tuning
Tuned Performance of OpenMP Programs on CUDA (1)

• Speedups are over serial on the CPU, when the largest available input data were used.
• Experimental Platform: CPU: two Dual-Core AMD Opteron at 3 GHz
  GPU: NVIDIA Quadro FX 5600 with 16 multiprocessors at 1.35GHz
Speedups are over serial on the CPU, when the largest available input data were used.

**Experimental Platform:**
- **CPU:** two Dual-Core AMD Opteron at 3 GHz
- **GPU:** NVIDIA Quadro FX 5600 with 16 multiprocessors at 1.35GHz
Lessons Learned

- General-Purpose GPUs (GPGPUs) do not perform well across a broad application range.
- Traditional caching optimizations may not work well on GPGPUs.
- Complex interaction between limited resources often requires tuning.
- Traditional profile-based tuning will not always work well on GPGPUs.
- Complex tiling transformations are not a cure-all solution.
Conclusion

- First compiler framework for automatic translation of OpenMP applications into CUDA-based GPGPU applications.
- Runtime tuning systems to address performance variation.
- New API, OpenMPC, providing improved programmability and tunability on CUDA.
- OpenMPC-based tuning system achieves performance improvements comparable to hand-coded CUDA.
Thank You!