Locality-sensitive Loop Scheduling in SOLLVE’s OpenMP

Vivek Kale, Brookhaven National Laboratory
SOLLVE: DoE’s fork of the LLVM OpenMP Implementation

- SOLLVE is a project to develop OpenMP for DoE exascale supercomputers.
- Can link it to your app through following http://github.com/SOLLVE/sollve
- Available on ECP Systems via Spack.

Our strategies are in runtime system and compiler in SOLLVE slab
#include <mpi.h>
int main(int argc, char** argv)
{
    MPI_Init(argc, argv);
    // input
    while (global_err < thresh)
    {
        MPI_Isend()/MPI_Irecv()/MPI_Waitall();

        for (i = 0; i < n; i++)
            doCompute(n);

        MPI_Collective_Op(&global_err);
        timestep++;
    }
    // output, viz
    MPI_Finalize();
}
#include <mpi.h>
#include <omp.h>

int main(int argc, char** argv)
{
    while (timestep < 1000)
    {
        #pragma omp parallel for schedule(static)
        for (i = 0; i < n; i++)
            loop_body(i);
        MPI_Op();
        timestep++;
    }
}

- Focus on the OpenMP computation region in an MPI+OpenMP program
  → Let’s use OpenMP’s dynamic loop schedule provided.
Hybrid Static/Dynamic Scheduling

- **Statically Scheduled Work**
- **Dynamically Scheduled Work**
- **Dequeue Overhead**
- **Thread barrier**

---

**Susceptible to imbalance.**

**Scheduler overhead stretches time.**

**Can reduce imbalance and sched ovhd. simultaneously.**

```c
#pragma omp parallel for schedule(static)
  for(int i=0; i<n; i++)
    loop_body(i);

#pragma omp parallel for schedule(static)
  for(int i=0; i<n; i++)
    loop_body(i);

#pragma omp parallel for nowait
  for(int i=0; i<n; i++)
    loop_body(i);

#pragma omp parallel for schedule(dynamic)
  for(int i=0; i<n; i++)
    loop_body(i);
```
Utility of Novel Strategies Shown

- Utility of novel strategies is demonstrated in published work by V. Kale et al.\textsuperscript{1,2} and others.
  - For example, mixed static-dynamic scheduling strategy with an adjustable static fraction.
    - To limit the overhead of dynamic scheduling, while handling imbalances, such as those due to noise.

CALU using static scheduling (top) and $f_d = 0.1$ (bottom) with 2-level block layout run on AMD Opteron 16 core node.

Diagram of static (top) and mixed static/dynamic scheduling (bottom) where $f_d$ is the dynamic fraction.
Enhancing Support in OpenMP to Improve Data Locality in Application Programs Using Task Scheduling

Martin Kong, Vivek Kale

Lower execution time due to lower L3 cache traffic volume.

Lower energy consumption due to less data movement.
AutoDock Mini-app

Created mini-app that allows us to study approaches to writing OpenMP-based GPU code, and later multi-GPU code

- Do T computations of floating-point vector multiplications, each of randomly chosen size between 0 and n, where n is an input to the program.
- **OpenMP CPU version**: OpenMP `parallel for` directive to parallelize T computations across CPU’s cores.
- **OpenMP GPU version**: Augment the CPU version by putting a `target` directive having `map` and `nowait` clauses inside the OpenMP `parallel for` to parallelize T computations across GPU’s SMs.

AutoDock mini-app with LLVM 11 on Summit shows that its task performance variation represents the behavior of full application.

OpenMP Task-to-multiGPU Scheduling

• Shared queue is used to represent GPUs and to manage access to them
• When a CPU acquires a GPU it launches its next task
• Several strategies used to map tasks to GPU

OpenMP CPU threads schedule an OpenMP task with a target region to a GPU (of a multi-GPU) that’s ready to run the task.
 inline unsigned gpu_scheduler_dyn(unsigned *occupancies, int ngpus) {
    short looking = 1;
    unsigned chosen;
    while (looking) {
        for (unsigned i = 0; i < ngpus; i++) {
            unsigned occ_i;
            #pragma omp atomic read
            occ_i = occupancies[i];
            if (occ_i == 0) {
                chosen = i;
                occupancies[chosen]++;
                looking = 0;
                break;
            }
        }
        #pragma omp target device(dev)
        map(to: a[0::arrSize], b[0::arrSize], c[0::arrSize])
        map(tofrom: success[i:1], output[i:1], taskWork[i:1],
            occupancies[dev:1])
        {
            devices[dev]++;
            if (taskWork[i] > probSize) taskWork[i] = probSize;
            const int NN = taskWork[i];
            output[i] = doWork(c, a, b, taskWork[i]);
            success[i] = 1;
        }
        #pragma omp task depend(in : success[i])
    }
    #pragma omp atomic read
    occupancies[dev]--;
}
return chosen;
}

#pragma omp parallel
#pragma omp single
#pragma omp taskloop shared(success)
for (int i = 0; i < numTasks; i++) {
    const int dev = gpu_scheduler_dyn(occupancies, ndevs);
    output[i] = 0;
    #pragma omp task depend(out : success[i])
    { success[i] = 0; }
    #pragma omp task depend(inout : success[i])
    {
        #pragma omp target device(dev)
        map(to: a[0::arrSize], b[0::arrSize], c[0::arrSize])
        map(tofrom: success[i:1], output[i:1], taskWork[i:1],
            occupancies[dev:1])
        {
            devices[dev]++;
            if (taskWork[i] > probSize) taskWork[i] = probSize;
            const int NN = taskWork[i];
            output[i] = doWork(c, a, b, taskWork[i]);
            success[i] = 1;
        }
        #pragma omp task depend(in : success[i])
    }
    #pragma omp atomic read
    occupancies[dev]--;
}
return chosen;
Results for Task-to-multi-GPU Strategies

- Ran AutoDock mini-app with uniformly random distribution, max task size 3400x3400.
- Compiled with LLVM 11 (with and without our patch) and executed on one node of Summit (42 CPU cores, 6 NVIDIA Tesla V100 GPUs, one thread per core).

- The MPI version of the AutoDock mini-app was used here for comparison.
- MPI (mpi-nopatch) gives 24.5x speedup over the CPU version with LLVM 11; static (sta-nopatch) gives 21.2x speedup over CPU version. MPI is 16.2% faster than this OpenMP node version.
- Round-robin (rrb) and random (ran) schedules partially alleviate load imbalance; dynmque provides lower-overhead scheduling.

→ Task-to-GPU scheduling techniques handle load imbalance, may reduce contention, and could be used to reduce data movement. Dynamic schedules improve performance 2X and more over MPI.
Proposal for User-defined Schedules in OpenMP

Example: glimpse of how a User-defined Schedule (UDS) might look like

```
typedef struct {...} schedule_data;
void myDynsStart(...) {}
void myDynsNext(...) {}
void myDynsFini(...) {}
#pragma omp declare schedule(myDyn) start(myDynsStart) next(myDynsNext) fini(myDynsFini)

void example() {
    static schedule_data sd;
    int chunkSize = 4;
    #pragma omp parallel for schedule(myDyn, chunkSize:&sd)
    for(int i = 0; i < n; i++)
        c[i] = a[i]*b[i];
}
```

- The directive `declare schedule` connects a schedule with a set of functions to initialize the schedule and hand out the next chunk of iterations.
- The syntax of the clause `schedule` is extended to also accept an identifier denoting the UDS.
- Instead of calling into the RTL for loop scheduling, the compiler will invoke the functions of the UDS.
- Visibility and namespaces of these identifiers will be borrowed from User-Defined Reductions in OpenMP 5.0.
An Implementation of the Static/Dynamic Schedule with UDS

Data Structures for the User-defined Scheduler

void mysd_start(int lb, int ub, int incr, int chunksz, loop_record_t *lr) {
    lr->lb = lb;
    lr->ub = ub;
    lr->incr = incr;
    lr->chunksz = chunksz;
    lr->counter = 0;
}

#define struct {
    int lb;
    int ub;
    int incr;
    int counter;
    double fs;
} loop_record_t;

void mysd_next(int *lower, int *upper, loop_record_t *lr) {
    int start;
    if(lr->counter < (lr->ub - lr->lb)*lr->fs*(lr->ub - lr->lb)/numthreads) {
        *lower = lr->fs*(lr->ub - lr->lb)*lr->counter;
        *upper = *lower + lr->chunksz*lr->incr;
    } else {
        #pragma omp atomic capture
        { start = lr->counter; lr->counter += lr->chunksz * lr->incr; }
        *lower = start;
        *upper = start + lr->chunksz * lr->incr;
    }
}

void mysd_fini(loop_record_t *lr) {
    // Do nothing
}

#pragma omp declare schedule(mysd) init(mysd_start)
next(mysd_next)
void example() {
    static loop_record_t lr;
    #pragma omp parallel for schedule(mysd, &lr)
    for (int i = 0; i < n; i++) {
        a[i] = s * a[i] * b[i];
    }
}

User-defined scheduler.

Application loop specifying a User-Defined Schedule
Extensions to OpenMP for Task-to-GPU Scheduling

- Previous results, showing need for load balancing along with data locality, motivate extensions to OpenMP.
- Such extensions should allow programmers to easily obtain application code performance through a locality-sensitive task-to-GPU load balancing.

Proposed Extension for OpenMP

Syntax:

```c
#pragma omp target scheduler clause[ [ [,] clause] ... ] new-line structured-block
```

where clause is one of the following:

- `num_devices(integer-expression)`
- `type( round_robin | dynamic | random | user_defined)` `affinity( temporal|spatial,)`
- `priority`

Example of Extension: miniAutoDock

```c
#pragma omp target scheduler num_devices(ndevs)
type(dynamic)
{
#pragma omp taskloop
  for(int i=0; i < numTasks; i++) {
    output[i] = 0;
  
#pragma omp target map(to: a[0:n*n], b[0:n*n], c[0:n*n]) map(tofrom: output[i:1], work) nowait
    {
      const int NN = n * n;
      double work_start = 0;
      for (int j = 0; j < NN; j++)
        c[j] = sqrt(a[j] * b[j]);
      output[i] = c[NN];
    } // end target
  } // end taskloop
} // end target
```
Lightweight Loop Scheduling in RAJA: lws-RAJA

Code through hand transformation or maybe ROSE/Orio/LLVM.

→ Significantly reduces lines of code for application programmer to use strategy: easy-to-use locality-sensitive scheduling strategies.

→ Improves portability of loop scheduling strategies.
Lassen is a front tracking code
Most of the computation is near the surface of the front.
Creates time-varying imbalances.

- Imbalance across nodes, and cores, have different dynamics as iterations progress.
- → Balancing both, in coordination, is necessary.
Load Balancing + Loop Scheduling Technique

Key Idea:

1. Modify across-node load balancing in Charm++ to assign load to one PE in each node.
2. Use my loop scheduling strategies in CkLoop to optimize within node performance.

This scheduler is merged into Charm++; Development history here: https://bitbucket.org/viveklkalew/ckloop_schedule/src/master/

Related Work

- DPLASMA, ParSec
- Legion
- Hybrid MPI+OpenMP
- Habanero
- OpenMP Guided Scheduling
- RAJA and Kokkos
Future Work

- Continue creating examples of user-defined schedules using LLVM OpenMP Implementation and experiment with them.
- Improvements to tasking for GPUs
  - Improved task scheduling strategies
  - Improve task affinity
- Add into OpenMP specification and LLVM OpenMP Implementation support for task-to-multiGPU scheduling.
- An eventual and longer term goal is to upstream the scheduling strategies into LLVM.
- We’ll also be looking at developing libraries for heterogeneous nodes that have intelligent runtime system support for handling load imbalance within-node, coordinating across-node load balancing with within-node load balancing.
- Finally, we’ll look into performance portability of loop scheduling in particular though use of RAJA.
Summary

- Load imbalance within node is an important problem
- Novel schedulers solve the problem
  - Basic static/dynamic scheduling
  - Variants of scheduling strategies
  - More work on task-to-multiGPU scheduling with GPUs
- Proposed extensibility features facilitate novel loop schedulers
  - OpenMP UDS
  - OpenMP task-to-GPU target scheduler
- Build on the scheduling strategies and make them accessible
  - UDS in RAJA → integration
  - Charm++ + CkLoop: → combination