

0.1 Project

1. Determine Parameters to Empirically Tune a Threaded Function
2. Develop Generic Threading Transformation in POET
3. Try Transformation on a Function
4. Empirically Tune and Test the Function
5. Analyze Results

0.2 POET Annotation of a Function

```

<parameter OS=1..3[2] "Operating_system , 1--Solaris , 2--Linux , 3--Mac"/>
<parameter M_Blocks=1..-[2] "Number_of_blocks_of_side_M"/>
<parameter N_Blocks=1..-[2] "Number_of_blocks_of_side_N"/>
<parameter K_Blocks=1..-[4] "Number_of_blocks_of_side_K"/>

<input gemmInput>
#include <cbblas.h>
/*@; BEGIN(gemm)
void dgemm( const enum CBLAS_TRANSPOSE TransA ,
            const enum CBLAS_TRANSPOSE TransB ,
            const int M, const int N, const int K,
            const double alpha , const double *A,
            const int lda , const double *B,
            const int ldb , const double beta ,
            double *C, const int ldc) //@=>gemmPrototype:FunctionDecl
{
    int m, n, k; //@;BEGIN(-)
    for (n = 0; n < N; n += 1) //@=>gemmDecl:Stmt
    { //@; BEGIN(body3)
        for (m = 0; m < M; m += 1) //@=>loopJ:Loop BEGIN(nest2)
        { //@;BEGIN(body2) BEGIN(parse)
            C[m+n*ldc] = beta * C[m+n*ldc]; //@ END(parse) =>_:Stmt
            for (k = 0; k < K; k +=1) //@=>loopL:Loop BEGIN(nest1)
            { //@;BEGIN(parse)
                C[m+n*ldc] += alpha * A[m+k*lda] * B[k+n*ldb]; //@END(parse) =>stmt1:
            } //@END(nest1:Nest) END(body2:Sequence)
        } //@END(nest2:Nest) END(body3:Nest)
    } //@END(nest3:Nest) END(gemmBody:Nest) END(-:S
}
/*@END(gemm:FunctionDefn) @*/
</input>

<trace gemmInput , gemm , gemmDecl , gemmBody , nest3 , loopJ , body3 , nest2 , loopI , body2 , nest1 ,

<define BuildDefaultThreadedFunctions DELAY
{
    body = BuildDefaultThreadedFunction(gemmPrototype , gemmBody , M_Blocks*N_Blocks*K_
    REPLACE(gemm , body , gemm);
}/>

<output "dgemm_default.c" (
    TRACE gemmInput ;
    APPLY BuildDefaultThreadedFunctions ;
    gemmInput
) />

```

0.3 POET Transformation

```
<* BuildDefaultThreadedFunction:
  Required inputs:
  fDecl – The original function’s prototype
  fBody – The original function’s complete body (less prototype)
  numThreads – The total number of threads you wish to split the function into

  This function will divide up the original function into numThreads pieces,
  and instantiate the serialized function numThread times.
*>

<xform BuildDefaultThreadedFunction pars=(fDecl ,fBody ,numThreads)
  iThreadStruct="ThreadData">
  if (fDecl : FunctionDecl#(TypeDecl#(funType ,funName) ,params))
  {
    structBody = BuildParameterStructBody(fDecl ,"" );
    ParameterStruct = DefaultParameterStruct(fDecl ,"" );
    SerialFuncLocalVars = DefaultSerialFuncVars(fDecl ,"" ,"" );
    SerialFunction = BuildSerialFunction(fDecl ,fBody ,SerialFuncLocalVars);

    ThreadedFuncLocalVars = DefaultThreadedFuncVars(fDecl ,fBody ,numThreads ,"" ,"" );
    ThreadedFuncLoopBody = BuildThreadedFuncLoopBody(fDecl ,"" ,numThreads ,"" );
    ThreadedFuncLoop = BuildThreadedFuncLoop(ThreadedFuncLoopBody ,0 ,numThreads);
    ThreadedJoinLoopBody = BuildThreadedJoinLoopBody("");
    ThreadedJoinLoop = BuildThreadedJoinLoop(ThreadedJoinLoopBody ,0 ,numThreads);

    Cleanup = BuildThreadedFuncCleanup(numThreads ,"" );

    ThreadedFunctionBody = BuildThreadedFuncBody(fDecl ,ThreadedFuncLocalVars ,"" ,
      ThreadedFuncLoop ,"" ,ThreadedJoinLoop ,"" ,Cleanup);
    ThreadedFunction = BuildThreadedFunction(fDecl ,ThreadedFunctionBody);

    Headers = Sequence#(HonorableMention(fDecl ,"" ),DefaultHeaders(""));
    body = Sequence#(Headers ,ParameterStruct);
    body = Sequence#(body ,SerialFunction);
    body = Sequence#(body ,ThreadedFunction);
    body
  }
  else
    ERROR(" expecting a function declaration" , " BuildDefaultThreadedFunction");
</xform>
```

0.4 Resulting Threaded Function

```
#include <blas.h>

/*****START POET THREADING*****/
/** The dgemm function was Threaded
    by the POET library
    Authored by Dr. Qing Yi UTSA
    Threaded functions implemented by Mike Stiles UTSA
**/

/**
 * Options Summary:
 *
 * Operating System - Linux
 * Madvise - Not Set
 * Number of Processors - 16
 * Processor Affinity - Not Set
 * Number of Processor Groups - 4
 * Memory Prefetch - Not Set
 * Number of blocks for side M - 2
 * Number of blocks for side N - 2
 * Number of blocks for side K - 1
 * Number of threads - 4
 * Order set for inner serial loops - N M K
 * Order set for outer serial loops - N M K
 * Order set for threaded loops - N M K
 * MB Factor - 44
 * NB Factor - 44
 * KB Factor - 44
**/

#include <assert.h>
#include <stdlib.h>
#include <pthread.h>

int Test_CpuMap[16] = {0,16,1,17,2,18,3,19,4,20,5,21,6,22,7,23};

/**
 * A struct derived from the
 * parameters listed in the original
 * serial function.
 * This struct will be used to pass
 * the parameters to the new serial
 * function from the new threaded function.
**/

typedef struct
```

```

{
enum CBLAS.TRANSPOSE TransA;
enum CBLAS.TRANSPOSE TransB;
int M;
int N;
int K;
double alpha;
double * A;
int lda;
double * B;
int ldb;
double beta;
double * C;
int ldc;;
int ThreadNum;
void* ThreadDataArray;
pthread_mutex_t *mutex;
} ThreadData;

void* dgemm_serial(void* local_ThreadData)
{
    /** Start local variables from function parameters **/
    enum CBLAS.TRANSPOSE TransA;
    enum CBLAS.TRANSPOSE TransB;
    int M;
    int N;
    int K;
    double alpha;
    double * A;
    int lda;
    double * B;
    int ldb;
    double beta;
    double * C;
    int ldc;
    /** End local variables from function parameters **/

    int m, n, k;
    int mB, nB, kB;
    ThreadData* local_ThreadData_ptr = (ThreadData*) local_ThreadData;
    int ThreadNum = local_ThreadData_ptr->ThreadNum;
    ThreadData* ThreadDataArray = (ThreadData*) local_ThreadData_ptr->ThreadDataArray;
    /** Start local variable assignments from function parameters **/
    TransA = local_ThreadData_ptr->TransA;
    TransB = local_ThreadData_ptr->TransB;
    M = local_ThreadData_ptr->M;
    N = local_ThreadData_ptr->N;
    K = local_ThreadData_ptr->K;
    alpha = local_ThreadData_ptr->alpha;
    A = local_ThreadData_ptr->A;

```

```

lda = local_ThreadData_ptr->lda;
B = local_ThreadData_ptr->B;
ldb = local_ThreadData_ptr->ldb;
beta = local_ThreadData_ptr->beta;
C = local_ThreadData_ptr->C;
ldc = local_ThreadData_ptr->ldc;
/** End local variable assignments from function parameters **/

```

```

for (n=0; n<N; n+=1)
{
  for (m=0; m<M; m+=1)
  {
    C[(m + (n * ldc))] = (beta * C[(m + (n * ldc))]);
    for (k=0; k<K; k+=1)
    {
      C[(m + (n * ldc))] = (C[(m + (n * ldc))] + ((alpha * A[(m + (k * lda))])));
    }
  }
}

```

```

void dgemm(const enum CBLAS_TRANSPOSE TransA ,
           const enum CBLAS_TRANSPOSE TransB ,
           const int M,
           const int N,
           const int K,
           const double alpha ,
           const double * A,
           const int lda ,
           const double * B,
           const int ldb ,
           const double beta ,
           double * C,
           const int ldc)

```

```

{
  register int i = 0;
  pthread_t* local_threads;
  ThreadData* local_ThreadData;
  register int M_Off;
  register int N_Off;
  register int K_Off;
  register int M_Rem;
  register int N_Rem;
  register int K_Rem;
  register int m;
  register int n;
  register int k;

```

```

M_Off = (((M / 44) / 2) * 44);

```

```

MRem = (M - (M_Off * 2));
N_Off = (((N / 44) / 2) * 44);
N_Rem = (N - (N_Off * 2));
K_Off = K;
K_Rem = 0;

local_threads = (pthread_t*) malloc((sizeof(pthread_t) * 4));
local_ThreadData = (ThreadData*) malloc((sizeof(ThreadData) * 4));
assert((local_threads && local_ThreadData));

for (n=0; n<2; n+=1)
{
    for (m=0; m<2; m+=1)
    {
        for (k=0; k<1; k+=1)
        {
            local_ThreadData[i].TransA = TransA;
            local_ThreadData[i].TransB = TransB;
            if ((m != 1))
            {
                local_ThreadData[i].M = M_Off;
            }
            else
            {
                local_ThreadData[i].M = (M_Off + MRem);
            }
            if ((n != 1))
            {
                local_ThreadData[i].N = N_Off;
            }
            else
            {
                local_ThreadData[i].N = (N_Off + NRem);
            }
            if ((k != 0))
            {
                local_ThreadData[i].K = K_Off;
            }
            else
            {
                local_ThreadData[i].K = (K_Off + KRem);
            }
            local_ThreadData[i].alpha = alpha;
            local_ThreadData[i].A = &A[((m * M_Off) + ((k * K_Off) * lda))];
            local_ThreadData[i].lda = lda;
            local_ThreadData[i].B = &B[((k * K_Off) + ((n * N_Off) * ldb))];
            local_ThreadData[i].ldb = ldb;
            if ((k == 0))
            {
                local_ThreadData[i].beta = beta;
            }
        }
    }
}

```

```

    }
    else
    {
        local_ThreadData[i].beta = 1;
    }
    local_ThreadData[i].C = &C[((m * M_Off) + ((n * N_Off) * ldc))];
    local_ThreadData[i].ldc = ldc;
    local_ThreadData[i].ThreadNum = i;
    pthread_create(&local_threads[i], NULL, dgemm_poet_test_serial , &local_Th
    i = (i + 1);
    }
}
}
for (i=0; i<4; i+=1)
{
    pthread_join(local_threads[i], NULL);
}
free(local_threads);
free(local_ThreadData);
}

```