Overlap Implementation for GWU-QCD Framework

Michael Lujan

Aug. 2010

The purpose of these notes is to explain how to construct the Dirac Overlap operator in the gwu-qcd framework.

1 Overview of Overlap

The massless overlap operator is given by

$$D_{ov}(0) = \rho(1 + \gamma_5 \epsilon(H)), \tag{1}$$

where $\rho = 4 - 1/2\kappa$, κ is a positive number. H, is the hermitian wilson operator, and $\epsilon(H)$ is the matrix sign function. The challenging part of computing $D_{ov}(0)$ is the construction of $\epsilon(H)$. $\epsilon(H)$ cannot be computed exactly everywhere, there is simply not enough computational resources available, and we therefore must approximate it. In this paper, we use a polynomial approximation which uses the Chebyshev polynomials as its basis. We will not get into explaining the mathematical theory behind this approximation. For further details see [1].

1.1 Deflation

In order to accelerate the computation of $\epsilon(H)$ we use deflation. The idea behind deflation is to use a small number of eigenvectors, $|\lambda\rangle$, and eigenvalues, λ, of H to compute the sign of the "small space" exactly. The Arnoldi algorithm allows us to compute a few of these eigenvectors numerically, typically about a 100 eigenvectors of H are used. We then may break up the the space of $\epsilon(H)$ as

$$\epsilon(H) = \epsilon(H)_{\text{small}} + \epsilon(H)_{\text{large}}. \tag{2}$$

The eigenvectors and eigenvalues of H allows us to compute the small space as follows

$$\epsilon(H)_{\text{small}} = \sum_{i=1}^{N} \text{sign}(\lambda_i) |\lambda_i\rangle\langle\lambda_i|$$

The large space then remains to be approximated via Chebyshev polynomials.

As a last note, to obtain the massive overlap operator, $D_{ov}(m)$, one uses the following relation

$$D_{ov}(m) = (\rho + m/2) + (\rho - m/2) D_{ov}(0) \tag{3}$$

2 Constructing Overlap in GWU-QCD Framework

The task in this section is to compute the following linear algebra equation using the gwu-qcd framework.

$$D_{ov}(0)\ \eta = \eta' \tag{4}$$

There are 3 different implementations of the Overlap operator: (1) GPU only, (2) CPU only, (3) mix GPU and CPU. Since the GPU only and CPU only construction are identical we'll show how to construct the GPU only overlap and the mix overlap.

2.1 Only on GPUs or on CPUs

In order to numerically construct overlap the user must input the necessary parameters. They are:

- Overlap kernel, which is H. H also encodes the information of ρ.
- Eigenvectors and eigenvalues of H. This is used to compute the small space.
- Precision of the approximation for the large space, usually 10^{-10}. This determines the accuracy and amount of matrix vector multiplications needed.

The overlap constructor is declared as

```
template<typename genvector>
overlap_polynomial<genvector>::overlap_polynomial(
    overlap_kernel<genvector>& kern,
    vec_eigen_pair<genvector>& kes, double max_prec)
```

The constructor is a template which allows for the overlap to take in either all GPUs or all CPUs implementation but not a mixture. Furthermore, the framework of gwu-qcd allows the vectors to also be either floats or doubles, this allows for roughly a factor of 2 in speedup when using conjugate gradient method to invert the overlap operator.

Once the overlap operator has been constructed there are only a handful of routines available for the user to access. In the following, genvecs are either CPU or GPU vectors.

- eps(genvec1, genvec2, error): This computes $\epsilon(H)$ on a vector.
- gamma5eps(genvec1, genvec2, error): This computes $\gamma_5\epsilon(H)$ on a vector.
- mult(genvec1, genvec2, mass, error): This computes the full massive overlap operator on a vector. i.e. $D_{ov}(m)$ on a vector.
- get_poly_order(error): This returns the order of the polynomial corresponding to the error or the precision requested.

The main routines, eps, gamma5eps, and mult, take in genvec1 as the source vector, and returns to the user the corresponding matrix vector multiplication in genvec2.

2.2 Mix GPUs and CPUs

The third implementation of overlap allows for a mixture of GPU and CPU vectors to be passed to the overlap constructor. Though ideally we would wish to keep all vectors on the GPU, because they are computationally faster, the relatively small amount of memory inhibits us from doing so. When this issue occurs we would have to allocate some vectors on the CPU. For this reason we have a mix overlap implementation. The constructor is varied slightly to be

```
overlap_polynomial_mix::overlap_polynomial_mix(overlap_kernel_mix& k,
    vec_eigen_pair<device_wilson_field>& kes,
    double max_prec, vec_eigen_pair<vector>* kes_cpu)
```

The mix overlap kernel is now differentiated than the previous one, and there are two vec eigen pairs, the first is for GPU vectors and the second is for CPU vectors, They must be passed in that order. The routines avaliable to the user are identical to the previous ones, however they are only for GPUs i.e. device_wilson_fields and not for CPU vectors.

2.3 Example

Here we give a concrete example of how to code the overlap operator and use a routine. This is shown for CPU vectors.

```
int main(void)
{
.
. //usual setup of links, bcs, etc.
.
//Constructor the eigen pair:hw_ieval is number of eigenpairs.
//desc is the lattice desc.
```

```
vec_eigen_pair<vector> eigen_pair(hw_ieval, desc);
read_vec_eigen_pair(hwevecname, hwevalname, eigen_pair);

//construct the overlap kernel: needs links and kappa
hwilson_cpu_kernel h_mult(links, kappa);

//set maximum precision
double prec = 1e-10;

//construct overlap called ov
overlap_polynomial<vector> ov(h_mult, eigen_pair, prec);

//do something with overlap
double mass = 0.12;
ov.mult(src,dest, mass);
.
.//you have just constructed the overlap operator.
.
}
```

References

[1] L. Giusti et. al., Comput.Phys.Commun. 153 (2003).